

How large RSA numbers are factored

Generated on 12 October 2026 by GPT-6

Introduction

In ordinary RSA, two large prime numbers are multiplied to make a public integer called the modulus. The primes are kept private. The factoring problem is to recover them from that public product.

With small numbers, the distinction is easy to see. Multiplying 47 by 53 gives 2,491. If someone proposes those factors, checking the answer takes one multiplication. If you are given only 2,491, you still need a method for discovering them.

Large RSA moduli pose the same kind of problem, but at a size where a straightforward search becomes impractical. Being able to perform enormous numbers of operations does not, by itself, make a poor search strategy useful. Before comparing programming languages, we need to understand what a good factoring algorithm asks the computer to do.

We will begin with the arithmetic behind factor extraction, then follow how a sieve collects the information that makes extraction possible. The implementation questions come afterward, when the different kinds of work inside the algorithm are visible.

Prerequisites

I assume basic programming and algebra. The specialized ideas below—modular arithmetic, smooth numbers, parity vectors, and large-integer representation—are explained here. If these are familiar, skim ahead.

A modulus is the number you are trying to factor

In ordinary two-prime RSA, the public modulus is

$$N = pq,$$

where p and q are primes. A *semiprime* is a product of two primes. Here we are interested in *balanced* semiprimes: the primes have comparable bit lengths. “Comparable” does not mean unusually close numerical values.

That distinction matters. If the primes are extremely close, a difference-of-squares search can be unusually effective. If one prime is small, an algorithm designed to find small factors may work well. Neither case is a fair stand-in for

a general RSA-style input. The standard algorithm taxonomy separates methods sensitive to the smaller factor from methods sensitive to the size of the whole number.¹

Names can also mislead. RSA-250 means **250 decimal digits**, or 829 bits. RSA-2048 means **2048 bits**, about 617 decimal digits. Those are very different problem sizes; the labels come from different naming conventions.²

Modular arithmetic keeps the remainder

The expression $a \equiv b \pmod{N}$ means that a and b leave the same remainder when divided by N . Equivalently, N divides their difference.

For example,

$$81 \equiv 4 \pmod{77}$$

because $81 - 4 = 77$. Arithmetic with these remainders lets us construct relationships involving N without already knowing its factors.

The greatest common divisor, written $\gcd(a, b)$, is the largest integer dividing both numbers. A GCD calculation is cheap compared with a large factorization. If we can construct a number sharing just one of N 's prime factors, the GCD extracts it.

Squares have even prime exponents

Consider

$$3600 = 2^4 \cdot 3^2 \cdot 5^2.$$

Every exponent is even, so the square root is $2^2 \cdot 3 \cdot 5 = 60$. For a positive integer, this is exactly the condition for being a square.

A number is *smooth* relative to a bound when all its prime factors are at most that bound. With a bound of 5, 3600 is smooth but 77 is not. A *factor base* is the collection of small primes we agree to track. Smooth numbers let us replace unwieldy integers with manageable lists of exponents. This is the essential bookkeeping behind sieve-based factoring.³

Parity turns multiplication into XOR

To determine whether a product is a square, we need only know whether each exponent is odd or even. Represent even by 0 and odd by 1. A list of these bits is a *parity vector*.

Using the primes 2, 3, and 5, take this deliberately small example:

Number	Factorization	Parity vector
6	$2 \cdot 3$	(1, 1, 0)
10	$2 \cdot 5$	(1, 0, 1)
15	$3 \cdot 5$	(0, 1, 1)

Multiplication adds exponents. Reducing those sums modulo 2 gives XOR: the three vectors XOR to zero. Indeed, $6 \cdot 10 \cdot 15 = 900 = 30^2$.

This example is arithmetic bookkeeping, not yet a factorization. We still need to connect the selected numbers to our modulus.

From equal squares to a large computation

The final step is surprisingly small

The goal of the sieve pipeline is to produce two squares with the same remainder:

$$x^2 \equiv y^2 \pmod{N}.$$

Subtracting and factoring gives

$$N \mid (x - y)(x + y).$$

The vertical bar means “divides.” If one prime factor of N divides the first bracket and the other divides the second, a GCD reveals them.

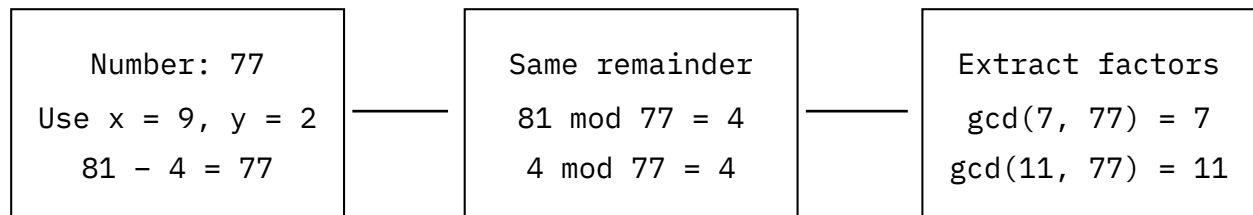
For $N = 77$, choose $x = 9$ and $y = 2$. Their squares differ by 77, so

$$\gcd(9 - 2, 77) = 7, \quad \gcd(9 + 2, 77) = 11.$$

We have factored 77. Try the demonstration below to see how the two roots split the factors.

A worked example

Step through the last mathematical move behind sieve-based factoring. These small pairs are supplied for you; finding useful pairs at RSA scale is the hard part.



1. The supplied square pair is 9 and 2. Their squares, 81 and 4, differ by 77.
2. Their difference and sum are 7 and 11. Both divide 77.
3. The check is $7 \times 11 = 77$. A pair with $x = y = 9$ fails: the gcds are 77 and 1, which reveal no new factor.

Figure 1. Extracting factors from a supplied congruence of squares. This example illustrates the final calculation, not the GNFS search for a useful pair.

Some congruences are useless. Choosing $x = 9, y = 9$ also makes the squares agree, but the GCDs are 77 and 1. Nothing has been split. Real algorithms may need another dependency before obtaining a nontrivial factor.¹

Instead of searching for one square, assemble it

The example with 77 begins with a useful pair already supplied. A factoring program has a harder task: it must find such a pair without knowing the factors of the input.

One approach is to look for several easier pieces of information and combine them. Each piece is a *relation*: an equation connecting a square modulo the input to a value whose small prime factors we know. The value need not be a square on its own. We keep its factorization so that we can later choose relations whose product is a square.

In the basic quadratic sieve picture, we examine values of

$$Q(t) = t^2 - N.$$

Here N is the number we are factoring, and t is a candidate integer. Subtracting N does not change the remainder modulo N , so the relation is

$$t^2 \equiv Q(t) \pmod{N}.$$

This connects two useful things. On the left, we already have a square and know its root, t . On the right, we have a number that we can try to factor over a small factor base. When that succeeds, we record the candidate and the exponents of its

prime factors. In this simplified example, we choose $t > \sqrt{N}$ so that $Q(t)$ is positive; implementations using negative values also track the sign.³

A complete small example

Take $N = 91$ and a factor base consisting of 2, 3, and 5. Suppose our search produces the following two candidates:

Candidate t	$Q(t) = t^2 - 91$	Factorization of $Q(t)$	Exponents for (2, 3, 5)
11	30	$2 \cdot 3 \cdot 5$	(1, 1, 1)
19	270	$2 \cdot 3^3 \cdot 5$	(1, 3, 1)

Neither 30 nor 270 is a square. Their exponent lists contain odd entries. But multiplying the two values adds the lists:

$$(1, 1, 1) + (1, 3, 1) = (2, 4, 2).$$

Now every entry is even. Consequently,

$$30 \cdot 270 = 2^2 \cdot 3^4 \cdot 5^2 = 90^2.$$

We have constructed the square on the right. We also know what happened on the left: the product of 11^2 and 19^2 is the square of $11 \cdot 19$. Multiplying the two recorded relations therefore gives

$$(11 \cdot 19)^2 \equiv 90^2 \pmod{91}.$$

The roots are $x = 209$ and $y = 90$. We can now use exactly the factor-extraction step from the demonstration:

$$\gcd(209 - 90, 91) = \gcd(119, 91) = 7,$$

$$\gcd(209 + 90, 91) = \gcd(299, 91) = 13.$$

The check is $7 \cdot 13 = 91$. We did not need either candidate value to be a square. We needed their known factorizations to combine into one.

This calculation also shows why the earlier parity vectors were useful. The exponent lists (1,1,1) and (1,3,1) both become (1,1,1) when reduced modulo 2. XORing them gives zero. The parity calculation identifies which relations to combine; we then return to the full exponents to construct the root 90.

For this tiny input, the two records can be chosen by inspection. With many primes and many relations, a program needs a systematic way to find a subset whose parity bits cancel. That is where the later linear-algebra stage enters. It solves a selection problem over the records we have collected, rather than searching directly for the unknown RSA primes.

What the sieve contributes

The example skips the work of obtaining the records. Most candidate values do not factor entirely over a small base. Repeatedly attempting a complete factorization of every value would waste effort on candidates we ultimately discard.

A sieve uses divisibility patterns to identify promising candidates. For example, in the ordinary integer sequence, every third position is divisible by 3. Polynomial values have their own patterns: once the program knows which residues of t make $Q(t)$ divisible by a prime, it can visit those positions repeatedly. This is a way to organize many divisibility checks together.

An ordinary prime sieve marks candidates for removal when it finds a divisor. A factoring sieve is interested in candidates with enough small divisors to account for their size. Finding evidence of divisibility is therefore encouraging. The promising values still need exact checking, and their complete exponent records must be correct.³

The output of relation collection is a body of checked arithmetic information. It is not yet a factorization of N . That distinction explains both the power and the cost of the method: the program gathers information that is relatively accessible, then performs another substantial computation to find the combination that makes it useful.

Different inputs call for different algorithms

Method	Where it belongs
Trial division / Pollard rho	Small factors and introductory experiments
Elliptic curve method (ECM)	Finding relatively small factors inside a much larger number
Quadratic sieve	Moderate general integers; useful for learning relation collection
GNFS	Very large general inputs, including balanced RSA moduli
Special number field sieve	Numbers with a favorable algebraic form; ordinary RSA moduli lack that advantage

These methods can cooperate. A factoring tool can remove small factors before committing to a sieve. ECM also appears inside sieve pipelines when candidate values have leftover factors. There is no universal digit cutoff: hardware, parameters, and input structure change the crossover.^{1, 4}

Why large RSA inputs use number fields

For sufficiently large general inputs, the main classical method is the **general number field sieve**, or GNFS. It preserves the “collect relations, cancel parities, obtain squares” strategy while using a more advantageous construction of the candidate values.

A number field extends rational-number arithmetic with a root of a polynomial. For a familiar example, adjoining $\sqrt{2}$ gives numbers of the form $a + b\sqrt{2}$ with rational a, b . GNFS uses a carefully chosen polynomial and a formal root, usually called α .⁵

In a simplified description, choose a polynomial f and an integer m satisfying $f(m) \equiv 0 \pmod{N}$. Algebraic expressions involving α can then be mapped into arithmetic modulo N by substituting m for α . The same candidate pair (a, b) contributes an ordinary value $a - bm$ and an algebraic value $a - b\alpha$.⁵

Why the algebraic side maps back to the modulus

The substitution may look arbitrary, so consider a small example using $N = 91$ again. Let

$$f(z) = z^2 - 30, \quad m = 11.$$

The polynomial's formal root α satisfies $\alpha^2 = 30$. The chosen integer satisfies the corresponding relationship modulo 91:

$$11^2 = 121 \equiv 30 \pmod{91}.$$

That agreement is what makes the substitution work. On the algebraic side, squaring $1 + \alpha$ gives $31 + 2\alpha$. Substituting 11 for α gives 53. On the ordinary side, squaring $1 + 11$ gives 144, whose remainder modulo 91 is also 53.

Both routes produce the same modular result. The algebraic arithmetic can be carried out in its own setting, then mapped back into arithmetic involving the number we want to factor. This example explains the compatibility of the two settings; choosing a polynomial that makes relation collection efficient is a further problem.

The two sides must stay linked throughout relation collection. It is not enough to find an ordinary smooth number in one search and an unrelated algebraic smooth value in another. The same candidate pair must provide compatible information on both sides. Otherwise the later products would not map to the same modular expression.

The algorithm searches for pairs that factor conveniently on both sides. The algebraic bookkeeping tracks prime ideals—objects that generalize prime-factor divisibility—and additional conditions. A square norm alone is insufficient to establish that an algebraic element is a square.⁶

The operational picture is two linked channels of factor information. Better candidate values produce useful relations more often. The deeper mathematics explains why those channels eventually map to equal squares.

The pipeline has several different bottlenecks

It helps to think of GNFS as a sequence of jobs with different outputs. The polynomial stage supplies a setup for the search. Sieving supplies relations. Filtering supplies a smaller system built from those relations. Linear algebra supplies selections of records. Square-root extraction turns a selection into a candidate factorization.

A later stage cannot simply compensate for arbitrary mistakes in an earlier one. An incorrect relation can spoil a dependency; an unnecessarily large or dense matrix can make the next stage more expensive. The program must preserve both correctness and a useful representation as information moves through the pipeline.

Polynomial selection chooses the setup. Spending more time here can improve later relation yield. It is a planning tradeoff: an extra hour selecting a polynomial might save more than an hour of sieving. The appropriate effort depends on the intended computation.

Relation collection searches the candidate space. Lattice sieving organizes this using work units associated with a chosen *special* q : a prime divisibility condition imposed on one side. A lattice here is a regularly structured set of integer pairs satisfying that condition. Restricting the search avoids testing arbitrary pairs without that known divisibility.⁷

Filtering turns collected relations into a usable system. Duplicates add no information. Relations involving a tracked factor that occurs nowhere else cannot participate in a parity cancellation. Removing and combining records reduces the later matrix, while choices about combinations affect how dense it becomes. This stage connects the amount of sieving to the cost of solving the system.²

Linear algebra finds dependencies in the parity matrix. Arithmetic is over $\text{GF}(2)$, the field containing just 0 and 1: addition is XOR, multiplication is AND. The matrix is *sparse*, meaning most entries are zero. A relation usually involves only a small part of the factor base, so its row needs relatively few entries. Filling every unused position into a dense grid wastes memory. It would also make the program process positions that contribute nothing.

This is a different workload from the original smoothness search. At this point, the arithmetic information already exists. The challenge is to move and combine a large quantity of binary data efficiently. That is why an improvement in relation collection does not automatically imply an improvement in the matrix stage.

Large implementations use methods such as block Wiedemann or block Lanczos. Block Wiedemann works through repeated sparse matrix-vector operations and relations within the sequences they generate. Its block form processes multiple vectors together; packed machine words can operate on many binary entries at once. This is exact bit arithmetic, not floating-point numerical approximation.⁸

Square-root extraction reconstructs the roots from selected relations. The algebraic side requires more than halving a list of ordinary prime exponents. After mapping the roots modulo N , compute the GCDs and verify the resulting factors. That short final calculation depends on all the information assembled earlier.⁶

01	02	03	04
Choose polynomials	Collect relations	Filter & solve	Extract factors
Make the values easier to factor.	Sieve many independent work units.	Find combinations with even exponents.	Build square congruences, then take gcds.

Figure 2. The GNFS pipeline. Relation collection can spread across many machines; the later matrix stage has different memory and communication needs.

The implementations already have substantial engineering

These are useful starting points, serving somewhat different purposes:

Project	What to study
CADO-NFS	A complete NFS pipeline; computational programs in C/C++, with orchestration scripts and network distribution
msieve	Factoring algorithms and NFS components, largely in C; repository documentation contains historical material, so check the exact revision
YAFU	Automated factoring and algorithm selection, largely in C; current builds bundle msieve and provide CPU-specific options and optional accelerator support
GMP	Efficient multiprecision arithmetic used underneath numerical software; it is an arithmetic library, not a complete factoring pipeline

CADO documents lattice sieving and an MPI-capable block Wiedemann implementation. YAFU’s current build documentation lists GMP and optional CUDA/OpenCL components. These capabilities describe the upstream projects; they should not be confused with a particular research team’s modified GPU port.^{4, 9, 10}

Large integers are stored as arrays of machine-sized pieces called *limbs*. An 829-bit integer needs thirteen 64-bit limbs; intermediate values may need more. Carries connect adjacent limbs. GMP includes assembly routines and switches multiplication algorithms with operand size, from ordinary multiplication to

Karatsuba, Toom methods, and FFT-based methods. Consequently, “rewrite multiplication” means competing with size-dependent algorithms and architecture-specific code.^{11, 12}

What GPUs and clusters change

Sieving can be divided into many work units with little coordination. Linear algebra has different constraints: repeated access to a large sparse matrix, substantial memory traffic, and communication when distributed. A machine configuration excellent for collecting relations may be less suitable for solving their matrix.

Recent results make the engineering opportunity concrete. Eric Lu’s September 2026 account reports factoring RSA-260 with GPU adaptations of CADO-NFS. He estimates 13.5 GPU-years of total work and describes irregular memory access as a central sieving challenge. Those are the author’s reported resource figures, not benchmarks independently reproduced for this article.¹³

Stephen Weis separately reports factoring RSA-896 on September 19, 2026, using a GPU port of CADO-NFS over ten calendar days and roughly 30 GPU-years. His note explicitly says the work did not meaningfully improve GNFS runtime. These milestones supersede the habit of treating the 2020 RSA-250 result as the latest example; neither demonstrates a new general factoring algorithm.¹⁴

A GPU-year and a CPU core-year are not interchangeable units of performance. Nor does ten days imply one GPU can do the job in ten days. Hardware count, utilization, communication, and the exact workload all matter.

Does Bend give you an advantage?

Bend is worth experimenting with, but use its **current version**, not an old description. As checked on October 12, 2026, the project is Bend 2. Its documentation says Bend 1 programs and HVM do not carry over. Current numeric types include Nat, U32, and F32, with no native U64, I64, or F64. Targets include C, Metal, CUDA, and JavaScript. It currently documents one GPU per program and no multi-machine execution.¹⁵

Its native parallel calls split independent work, but the guide requires roughly balanced execution times. The current scheduler does not move an assigned task between cores. That makes variable-duration work an important experiment rather than a detail to ignore.¹⁶

My assessment: Bend could make a suitably shaped kernel easier to express and explore. It does not automatically solve the memory-access problem in sieving, provide a mature large-integer library, or coordinate an NFS cluster. You would need to investigate representation, foreign interfaces, scheduling, and exact

arithmetic. Native numeric support is not the same as an inability to represent bigger numbers: you can build a multi-limb representation, but implementing it efficiently is additional work.

An interesting first comparison is a fixed-size batch of modular arithmetic or candidate scoring, with the same inputs and arithmetic widths in Bend and a well-optimized baseline. If Bend loses, determine whether the cause is representation, compilation, task overhead, or memory behavior. If it wins, integrate that kernel and measure whether the whole pipeline improves.

Should you go down to assembly?

Only after measurement points there. GMP already uses assembly extensively. Its own guidance says low-level routines matter especially at small and moderate sizes, while algorithm selection becomes more important at larger sizes.¹⁷

Suppose a routine accounts for 20% of total runtime. Making it twice as fast changes total runtime from 1 unit to $0.8 + 0.1 = 0.9$ units: an overall speedup of about 1.11 times. Even deleting that routine entirely cannot exceed 1.25 times. This is Amdahl's law, derived directly from the time breakdown.

Before asking AI to generate assembly, ask it to explain the profile, existing instructions, data dependencies, and measurement setup. A changed memory layout or batch boundary may save more than a locally clever instruction sequence. Exact arithmetic also needs checks for carries, overflow, boundaries, and exceptional inputs. Passing a few random examples is insufficient evidence.

Further work

Large factoring is a pipeline for constructing a useful congruence of squares. Better engineering can reduce its cost; a new algorithm would change how that cost grows. Those are different achievements, and both are worthwhile.

GNFS has a heuristic subexponential running-time estimate: better growth than an exponential function of the input's bit length, but worse than a polynomial. "Heuristic" means the usual analysis relies on assumptions about the behavior of the values generated, rather than proving every feature of the practical algorithm. There are rigorous analyses of modified randomized NFS constructions, which are a separate matter from certifying a production implementation.⁵

There is no proof that we have reached the best possible classical algorithm or implementation. That leaves room for research; it does not establish that a particular rewrite will succeed. Distinguish a faster kernel, a cheaper complete computation, and a better growth rate before deciding what you want to improve.

For your side project, I would proceed in this order:

1. **Build understanding with small inputs.** Implement GCD, modular multiplication, and a toy quadratic sieve. Verify every reported factorization by multiplication. This is learning code, not a performance claim.
2. **Establish a baseline.** Run an existing tool on several generated balanced semiprimes, initially around 30–60 decimal digits. Keep inputs, factor-generation method, versions, compiler options, hardware, time, and memory use. Move upward only after observing the cost.
3. **Study NFS stage timings.** Use CADO on a manageable larger input. Choose one stage or kernel with a visible cost. Small-input results alone do not establish behavior at record sizes.
4. **Compare implementations fairly.** Keep arithmetic, workloads, and output checks identical. Include data movement and setup costs where they occur in real use. Separate latency, throughput, resource consumption, and total factoring time.
5. **Try Bend as a measured branch.** Give it a bounded kernel with enough independent work. Compare against optimized C/C++ or CUDA on the same hardware before considering a wider rewrite.

No benchmark was run for this article, so no Bend speedup is claimed. A good first result might be an accurate, reproducible explanation of why one implementation loses. A stronger result might be a modest improvement that survives varied inputs and makes the real pipeline cheaper.

For deeper study, follow three threads: quadratic sieving to understand relation collection, sparse binary linear algebra to understand dependency extraction, and profiling one actual NFS kernel to connect the mathematics to hardware.

References

1. Menezes, van Oorschot, and Vanstone, Handbook of Applied Cryptography, chapter 3. <https://cacr.uwaterloo.ca/hac/about/chap3.pdf>.
2. Boudot and colleagues, Comparing the difficulty of factorization and discrete logarithm. <https://arxiv.org/abs/2006.06197>.
3. Carl Pomerance, Smooth numbers and the quadratic sieve. <https://websites.math.leidenuniv.nl/algebra/sieving.pdf>.
4. CADO-NFS. <https://cado-nfs.gitlabpages.inria.fr/>.
5. Number Field Sieve with Provable Complexity. <https://arxiv.org/abs/2007.02689>. for the distinction between heuristic and rigorous complexity analysis.
6. Peter Stevenhagen, The number field sieve. <https://pub.math.leidenuniv.nl/~stevenhagenp/ANTproc/old/04psh.pdf>.
7. McGuire and Robinson, A New Angle on Lattice Sieving. <https://arxiv.org/abs/2001.10860>.
8. CWI, Finding column dependencies by block Wiedemann. <https://ir.cwi.nl/pub/4642/>.
9. msieve. <https://github.com/radii/msieve>.

10. YAFU. <https://github.com/bbuhrow/yafu>.
11. GMP: low-level functions. https://gmplib.org/manual/Low_002dlevel-Functions.
12. GMP: multiplication algorithms. <https://gmplib.org/manual/Multiplication-Algorithms>.
13. Eric Lu, Factoring RSA-260. <https://cognition.com/blog/factoring-rsa-260>. September 2026.
14. Stephen Weis, RSA-896. <https://saweis.net/posts/rsa-896.html>. September 2026.
15. Bend 2 repository.
<https://github.com/bendlang/bend/blob/d71f42ee287bd7b9f86b44df44b210ac5eb651bd/README.md>. checked October 12, 2026, at revision d71f42ee287b (October 11). Pin this or another stated revision before reproducing an experiment.
16. Bend 2 guide.
<https://github.com/bendlang/bend/blob/d71f42ee287bd7b9f86b44df44b210ac5eb651bd/guide/GUIDE.md>. checked October 12, 2026, at revision d71f42ee287b (October 11). Pin this or another stated revision before reproducing an experiment.
17. GMP: assembly coding. <https://gmplib.org/manual/Assembly-Coding>.