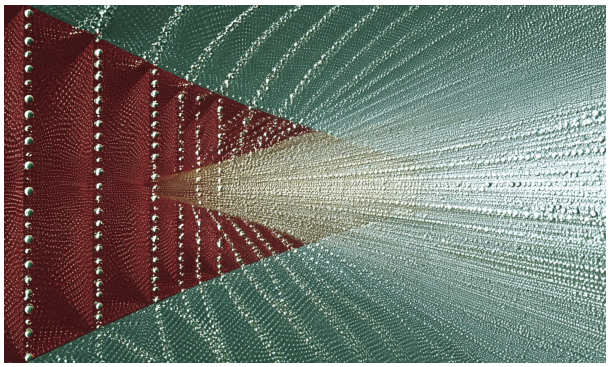
Factoring Integers After The Dark Ages

Stephen J. Mildenhall

2026-02-27



I recently created a webpage to factor integers. I remember a BASIC program to factor integers was one of my first programs. It was cool to learn that 1001 = 7 x 11 x 13. Experimenting with modern factoring routines blew my mind, so fast for such large numbers. But then math-me asked: is it actually that amazing? How hard is a random integer to factor, as opposed to an RSA product of two equal sized primes? How many factors do you expect? What sizes? A bit of exploration convinced me that modern factoring methods are actually more amazing than I initially thought. There's virtually no trial division. But a central place for the birthday collision problem and modular arithmetic. There are eight levels of magic, involving lots of cool number theory. Including my friends Elliptic Curves—subject of my PhD thesis. This LLM-fueled post describes the magic and how the open-source program YAFU (Yet Another Factoring Utility) weaves then together to deliver spectacular results.

Expectations of a Random Integer

Before we can master the "How" of factorization, we must understand the "What." When we reach into the infinite bag of integers and pull one out, we aren't just grabbing a needle in a haystack; we are grabbing a structured object governed by the laws of probabilistic number theory.

The Density of Primes

The journey begins with a binary question: Is it prime? The **Prime Number Theorem (PNT)** tells us that the probability an integer X chosen uniformly at random up to N is prime is $\approx 1/\ln N$. For an n-digit number ($N = 10^n$), this probability scales linearly

with the number of digits:

$$P(X \text{ is prime}) \approx \frac{1}{\ln(10^n)} = \frac{1}{n \ln 10} \approx \frac{0.434}{n}$$

For a 100-digit number, you'll find a prime roughly every 230 integers. This is the first wall any factorization utility hits. If you don't check for primality first, you might spend a billion years trying to find factors that don't exist.

How Many Factors? The Erdős–Kac Theorem

If the number isn't prime, how many factors should we expect? A common intuition is that a 100-digit number must have dozens of prime factors. The truth, provided by the **Erdős–Kac Theorem**, is far more sparse.

Erdős–Kac is essentially the Central Limit Theorem for prime factors. It states that the number of distinct prime factors $\omega(X)$ of a random integer $X \leq N$ follows a normal distribution:

$$\frac{\omega(X) - \ln \ln N}{\sqrt{\ln \ln N}} \sim \mathcal{N}(0, 1)$$

This leads to a startling realization: the average number of prime factors grows with the *logarithm of the number of digits*. For a 100-digit number: $N = 10^{100}$, $\ln N \approx 230.25$ and $\ln \ln N \approx 5.44$. The average 100-digit number has only **5 or 6 distinct prime factors**.

Precision Counting: The Sathe-Selberg Formula

Erdős–Kac is an asymptotic result. For precise probabilities of having *exactly* k factors, we turn to the **Sathe-Selberg formula**. This formula allows us to estimate the count of k-factor integers $\pi_k(x)$ as:

$$\pi_k(x) \sim \frac{x}{\ln x} \frac{(\ln \ln x)^{k-1}}{(k-1)!} G\left(\frac{k-1}{\ln \ln x}\right)$$

where $G(z)$ is an analytic function defined by:

$$G(z) = \frac{1}{\Gamma(z+1)} \prod_p \left(1 + \frac{z}{p-1}\right) \left(1 - \frac{1}{p}\right)^z$$

$G(z)$ is slowly varying. If $k \approx \ln \ln N$ (the average case), then $z \approx 1$ and $G(1) = 1$. G matters in the tail of the distribution.

For small k, this formula reveals that the distribution is **Poisson-like**. As k moves far from $\ln \ln N$, the density of drops off exponentially. Why does the distribution of prime factors look so much like a Poisson distribution?

If you view the existence of each prime factor as an independent event with probability $1/p$, then the expected number of factors for $X \leq N$ is the sum of these probabilities:

$$\sum_{p \leq N} \frac{1}{p} \approx \ln \ln N$$

This $\lambda = \ln\ln N$ becomes the intensity of our factor-finding process. The **Sathe-Selberg formula** is essentially a Poisson distribution with a corrective number-theoretic layer $G(z)$:

$$\Pr = \frac{\pi_k(x)}{x} \approx \underbrace{\left(\frac{(\ln\ln x)^{k-1}}{(k-1)!\ln x}\right)}_{\text{Poisson Component}} \times \underbrace{G\left(\frac{k-1}{\ln\ln x}\right)}_{\text{Correction Factor}}.$$

As long as you are investigating numbers near the average—the typical random integer—then $G(z) \approx 1$, and you can rely on the Poisson approximation to give you accurate probabilities. It is only in the tails of the distribution (like seeking primes or numbers with an extreme abundance of factors) where the prime product $G(z)$ exerts its influence.

The Poisson approximation of Sathe-Selberg for the density of k-factor integers writes

$$\text{Density} \approx \frac{1}{\ln N}\frac{(\ln\ln N)^{k-1}}{(k-1)!}$$

with $\lambda = \ln\ln N$ and $m = k-1$. The Poisson formula gives

$$P(k-1) = \frac{(\ln\ln N)^{k-1}e^{-\ln\ln N}}{(k-1)!}$$

Since $e^{-\ln\ln N} = \frac{1}{\ln N}$, the Poisson probability for $k-1$ events is identical to the leading term of the Sathe-Selberg formula.

```
import numpy as np
from scipy.special import gamma, factorial

def G_factor(z, prime_limit=10000):
    """
    Computes the Sathe-Selberg correction factor G(z).
    z = (k-1) / ln(ln(N))
    """
    if z == 1: return 1.0 # The average case simplification

    # Simple Euler product over first 'prime_limit' primes
    res = 1.0 / gamma(z + 1)
    # Note: In a real implementation, you'd use a pre-computed prime list
    # and a more sophisticated product handling for convergence
    for p in primes_up_to(prime_limit):
        term = (1 + z/(p-1)) * ((1 - 1/p)**z)
        res *= term
    return res

def sathe_selberg_pi_k(N, k):
    ln_N = np.log(N)
    ln_ln_N = np.log(ln_N)

    z = (k - 1) / ln_ln_N
```

```
    poisson_term = (1 / ln_N) * (ln_ln_N**(k-1)) / factorial(k-1)

    return poisson_term * G_factor(z)
```

Smoothness and the Dickman-de Bruijn Function

In the factorization business, the most important property is **smoothness**. We say an integer is **y-smooth** if all its prime factors are $\leq y$. The probability that a random number is y-smooth is given by the **Dickman function** $\rho(u)$, where $u = \ln X / \ln y$ is the ratio of total digits to smooth digits.

The function is defined by the delay-differential equation:

$$u\rho'(u) + \rho(u-1) = 0, \text{ with } \rho(u) = 1 \text{ for } 0 \leq u \leq 1$$

As u increases, the probability of smoothness plummets.

- For small u (e.g., $u = 2$): Roughly 30% of numbers are $N^{1/2}$-smooth. These are the balanced numbers (like RSA keys) and they are relatively common.
- Large u (e.g., $u = 10$): Only 1 in 10^{13} numbers are $N^{1/10}$-smooth. If you're looking for a 10-digit factor in a 100-digit number, you are fighting these odds.

The Largest Prime Factor: Golomb–Dickman

If we pick a random number and factor it completely, what is the size of the boss—the largest prime factor $P_1(X)$?

The expected fraction of digits of the largest prime factor relative to the total digits of N is the **Golomb–Dickman constant**:

$$G = \int_0^\infty \rho(u)du \approx 0.624329...$$

This means that for your random 100-digit number, you should expect the largest prime factor to be roughly **62 digits long**. This constant is a fundamental limit on why we need sophisticated sieving like QS or GNFS—because trial division will never reach that 62-digit titan.

The Decay Table

To summarize the landscape, here is how the probability of finding a $\leq B$-smooth 100-digit number N decays as B gets smaller. This is the probability that none of the prime factors of N exceed the bound B.

u	B (Smoothness)	$\rho(u) \approx$	The Factorizer's Reality
1	10^{100}	1.0	Every number is N-smooth.

u	B (Smoothness)	$\rho(u) \approx$	The Factorizer's Reality
2	10^{50}	0.306	Balanced semiprimes live here. Hard for Pollard.
3	10^{33}	0.048	ECM starts to excel here.
4	10^{25}	0.0049	Pollard's Rho finds these in minutes.
6	10^{16}	1.9×10^{-5}	Rare easy numbers; Rho finds these in seconds.
10	10^{10}	2.7×10^{-13}	Virtually impossible to be this smooth.

Appreciating this table is the transition between Dark Ages guesswork and modern algorithmic strategy.

The Dark Ages and the Foundation of Magic

Before we can appreciate the Magic of sub-exponential algorithms, we must understand why the old ways fail and the mathematical bedrock upon which modern factorizers are built.

The Complexity of the Dark Ages

The most primitive method of factorization is **Trial Division**: checking every integer $d \in [2, \sqrt{N}]$ to see if d divides N. (Or every d so that $d < N/d$ to avoid computing an "expensive" square-root as my 1970s CS teacher explained!)

The complexity is the tragedy here. If N has n digits, trial division is $O(\sqrt{10^n}) = O(10^{n/2})$. This is **exponential complexity** in terms of the input size n.

Consider a 100-digit RSA-style semiprime: $\sqrt{N} = 10^{50}$. Even if your computer could perform 10^{15} divisions per second (a massive supercomputer cluster), it would take $\approx 3.17 \times 10^{27}$ years to find the factor.

The Dark Ages aren't just slow; they are physically impossible for the numbers that secure our modern world.

Magic #0: The Greatest Common Divisor (GCD)

If there is a patron saint of factorization, it is Euclid. The **Euclidean Algorithm** is Magic #0 because it allows us to find the shared factors between two numbers without factoring either of them.

The algorithm relies on the observation that $\gcd(a, b) = \gcd(b, a \pmod b)$. By repeatedly applying this, we reduce the problem to a series of remainders.

Why it is Magic: The complexity of GCD is $O((\ln N)^2)$, which is **polynomial time** in the number of digits. It is blazingly fast. In every advanced algorithm (Pollard's ρ, ECM, QS), the final step is almost always a GCD. We use complex math to generate a number X that we *hope* shares a factor with N, and then we let Euclid do the final pluck.

Magic #1: Binary Exponentiation

Many primality tests and factorization steps require us to calculate $a^E \pmod N$, where E might be as large as N itself. Calculating this by multiplying a by itself E times would take us right back to the Dark Ages.

The Magic: We use the binary representation of E.

1. **Square:** Generate a sequence $a^1, a^2, a^4, a^8, a^{16} \ldots \pmod N$ by repeatedly squaring the previous result.
2. **Multiply:** Combine only the powers where the corresponding bit in E is 1.

For a 100-digit E, we only need ≈ 333 squarings and at most 333 multiplications. This reduces a lifetime of the universe calculation to a few microseconds. Without binary exponentiation, even the simplest primality test would be unusable.

Primality Testing

Before we spend a single CPU cycle trying to find a factor, we must ensure the number isn't prime. In the modern era, we don't look for factors to prove primality; we look for witnesses to compositeness.

Fermat's Little Theorem

Fermat gave us the first great Gatekeeper: If p is prime, then for any $1 < a < p$:

$$a^{p-1} \equiv 1 \pmod p$$

If we find an a such that $a^{N-1} \not\equiv 1 \pmod N$, then N is **definitely composite**. If the congruence holds, N is a probable prime.

The Flaw: Carmichael numbers (like 561). These composite numbers satisfy the Fermat test for *all* bases a. We need a stronger gatekeeper.

Magic #2: The Miller-Rabin Test

Miller-Rabin refines Fermat's idea by exploiting the fact that in a prime field, the only square roots of 1 are 1 and -1. (Consider $N = 15$. $4^2 = 16 \equiv 1 \pmod{15}$. Here, 4 is a square root of 1 that is neither 1 nor -1. This is a non-trivial square root and shows N is not prime.)

For an odd N, we write $N - 1 = 2^s \cdot d$. We then look at the sequence:

$$a^d, a^{2d}, a^{4d}, \ldots, a^{2^s d} \pmod N$$

If N is prime, this sequence must either start with 1, or it must hit -1 $(N-1)$ at some point before it reaches 1. If it reaches 1 without having hit -1 just before it, we have found a **non-trivial square root of 1**, which is a mathematical proof that N is composite.

Example 0.1 (Miller-Rabin in Python)**.** This implementation uses `pow(a, d, n)` (which uses Binary Exponentiation under the hood) and returns a `pandas` DataFrame to visualize the witness process.

```
import random
import pandas as pd

def miller_rabin(n, k=5, verbose=False):
    """
    Industrial-grade Probabilistic Primality Test.
    k = number of rounds (bases to test)
    """
    if n < 2: return False, None
    if n == 2 or n == 3: return True, None
    if n % 2 == 0: return False, None

    # Step 1: Write n-1 as 2^s * d
    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1

    history = []

    for i in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)

        iteration_data = {
            "iteration": i + 1,
            "base_a": a,
            "initial_x": x,
            "outcome": "Probable Prime" if x == 1 or x == n-1 else "Checking S-steps"
        }

        if x == 1 or x == n - 1:
            if verbose: history.append(iteration_data)
            continue

        # Repeat squaring to find -1
        for r in range(1, s):
            x = pow(x, 2, n)
            if x == n - 1:
                iteration_data["outcome"] = f"Probable Prime (found -1 at s={r})"
                break
        else:
            # If we never hit -1, it's definitely composite
```

```
            iteration_data["outcome"] = "COMPOSITE"
            if verbose:
                history.append(iteration_data)
                return False, pd.DataFrame(history)
            return False, None

        if verbose: history.append(iteration_data)

    return True, pd.DataFrame(history) if verbose else None
```

Magic #3: Pollard's Rho or The Birthday Problem at Work

If the number passes our primality tests and is confirmed composite, we don't jump straight to the heavy sieving algorithms. We first try to catch mid-sized factors using the Birthday Problem logic of **Pollard's ρ method**.

The core intuition is the collision trick, aka Birthday problem. Imagine you are in a room with N people. To find someone with a specific birthday (say, January 1st), you need to ask nearly 365 people. This is **Trial Division** logic—it's linear.

But to find *any* two people who share a birthday, you only need 23 people. This is the **Birthday Problem**. In factorization, we don't look for a factor p; we look for two numbers x and y such that $x \equiv y \pmod{p}$.

The algorithm uses a random walk in a small ring: collisions are inevitable. We use a pseudorandom function, usually $f(x) = (x^2 + c) \pmod{N}$, to generate a sequence. 1. The sequence is calculated modulo N (the big number). 2. However, it effectively shadows a sequence modulo p (the smaller, unknown factor). 3. Because $p < N$, a collision $x_i \equiv x_j \pmod{p}$ will happen much faster—specifically in $O(\sqrt{p})$ steps.

Once we have a collision modulo p, then p divides the difference $|x_i - x_j|$. We use **Magic #0 (GCD)** to extract it:

$$g = \gcd(|x_i - x_j|, N)$$

To find this collision without storing every x in memory, we use two pointers in a method called Floyd's circle-finding. The **Tortoise** moves one step at a time, and the **Hare** moves two steps. In a finite set, the sequence eventually enters a cycle, forming the shape of the Greek letter ρ. The Hare will eventually lap the Tortoise inside the cycle modulo p.

Example 0.2 (Pollard's Rho in Python)**.**

```
import math

def pollard_rho(n, c=1):
    """
    Pollard's Rho algorithm with Floyd's cycle-finding.
    Returns a factor of n.
    """
    if n % 2 == 0: return 2
```

```
    def f(x): return (x*x + c) % n

    x = 2; y = 2; d = 1

    while d == 1:
        x = f(x)           # Tortoise moves 1 step
        y = f(f(y))        # Hare moves 2 steps
        d = math.gcd(abs(x - y), n)

        if d == n:
            # Failure: hit a collision mod n, try a different constant c
            return pollard_rho(n, c + 1)

    return d
```

The Power: For a factor $p \approx 10^{14}$, trial division needs 10^{14} steps. Pollard's ρ needs $\approx \sqrt{10^{14}} = 10^7$ steps. On a modern CPU, this is instantaneous.

Magic #4: Pollard's $p-1$ Method

If Pollard's ρ is a random walk, the $p-1$ method is a targeted strike. It is incredibly fast, but it has a very specific weakness: it only works if the number N has a prime factor p such that the number $p-1$ is **smooth**.

Recall **Fermat's Little Theorem**: If p is prime, then for any a not divisible by p:

$$a^{p-1} \equiv 1 \pmod{p}$$

Now, suppose p is a factor of our composite N. If $p-1$ is smooth, it means $p-1$ is composed of small primes. Therefore, $p-1$ will divide some large factorial $B!$ (or a product of small prime powers). Let's call this large number K.

If $(p-1) \mid K$, then by Fermat's theorem:

$$a^K = a^{(p-1)\cdot \text{something}} = (a^{p-1})^{\text{something}} \equiv 1^{\text{something}} \equiv 1 \pmod{p}$$

If $a^K \equiv 1 \pmod{p}$, then p divides the difference $(a^K - 1)$. Since p also divides N, we can use our favorite foundation, **Magic #0 (GCD)**, to find it:

$$g = \gcd(a^K - 1, N)$$

If g is between 1 and N, we have successfully plucked the prime p out of N.

The $p-1$ method is a one-shot deal for a specific factor p.

- **If $p-1$ is smooth:** The algorithm finds p almost instantly using binary exponentiation and one GCD.
- **If $p-1$ is NOT smooth:** The algorithm fails completely. There is no way to tweak it to make it work for that specific p.

Pollard's method is extended by using elliptic curves where we aren't stuck with the fixed value group size $p-1$. By changing the elliptic curve, we effectively change the target from $p-1$ to a random group order g in the Hasse interval $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$. If $p-1$ isn't smooth, maybe g is!

Example 0.3 (Python Implementation: Pollard's $p-1$.)**.** Here is a simple version that uses a factorial-based K.

```
import math

def pollard_p_minus_1(n, B=10000):
    """
    Attempts to find a factor of n using Pollard's p-1 method.
    B is the smoothness bound.
    """
    a = 2 # Usually start with base 2

    # Compute a^(B!) mod n efficiently
    # Instead of computing B!, we just do successive powers
    for j in range(2, B + 1):
        a = pow(a, j, n)

    g = math.gcd(a - 1, n)

    if 1 < g < n:
        return g  # Success!
    else:
        return None # Failure: p-1 was not B-smooth
```

Magic #5: The Elliptic Curve Method (ECM)

When Pollard's ρ reaches its limit (usually factors around 15–20 digits), we bring in ECM. Invented by Hendrik Lenstra, ECM is the first algorithm whose complexity depends on the size of the smallest factor p rather than the size of N, but it does so in a way that is much more flexible than ρ. ECM is a generalization of Pollard's $p-1$ algorithm. In the $p-1$ method, we hope that $p-1$ is smooth (composed of small primes). If it isn't, the algorithm fails. In ECM, we replace the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^\times$ with the group of points on an elliptic curve E over $\mathbb{F}_p$.

- The order (number of points) of the group is $g = |E(\mathbb{F}_p)|$.
- By Hasse's Theorem, this order is always in the range $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$.

The magic is that if one curve doesn't have a smooth order, we simply pick a new curve with different parameters a and b. Each curve gives us a new chance to hit a smooth group order.

1. Pick a random curve E and a point P on it.
2. Attempt to compute $Q = kP$, where k is a product of many small primes.
3. In the process of point addition, we must perform a modular inverse. This inverse is calculated using the Extended Euclidean Algorithm.
4. If the inverse fails because the number isn't coprime to N, the GCD will suddenly reveal the factor p.

While ECM is primarily a factorizer, it is also the basis for ECPP (Elliptic Curve Primality Proving).

- To prove p is prime, we find a curve whose order m has a large prime factor $q > (\sqrt{p}+1)^2$.
- If we can show a point P has order q in $E(\mathbb{F}_p)$, then p is prime.
- This is recursive: we then have to prove q is prime, and so on, until we hit a small prime.

ECM is an excellent mid-game strategy. You can run hundreds or thousands of curves (curvesets) to find factors up to 40 or 50 digits before finally giving up and moving to the boss level: the Sieve.

Details

An elliptic curve is defined by the Weierstrass equation:

$$y^2 = x^3 + ax + b$$

The parameters a and b are coefficients chosen from $\mathbb{Z}/N\mathbb{Z}$. For the curve to be non-singular (smooth), we require the discriminant $\Delta = 4a^3 + 27b^2 \not\equiv 0 \pmod{N}$.

If p is a prime divisor of N, then any calculation we do modulo N is also happening hidden modulo p. There is a natural projection (homomorphism):

$$\phi : E(\mathbb{Z}/N\mathbb{Z}) \to E(\mathbb{F}_p).$$

We used this in Pollard's ρ method too.

When we add points on a curve modulo N, we are simultaneously adding points on a shadow curve over the finite field $\mathbb{F}_p$. The number of points on this shadow curve is the **group order** $g = |E(\mathbb{F}_p)|$. By **Hasse's Theorem**, this order g is always near p:

$$g \in [p + 1 - 2\sqrt{p},\ p + 1 + 2\sqrt{p}]$$

ECM is a generalization of **Pollard's $p-1$ algorithm**. In the $p-1$ method, we need the integer $p-1$ to be smooth (composed of small primes). If it isn't, the algorithm is stuck. The Magic of ECM is to find a smooth group order g for our chosen curve E. If it isn't, we don't give up—we just **change a and b** to create a completely different curve with a new group order g'. Because g' is a different random integer in the Hasse interval, we get a fresh roll of the dice to hit a smooth number.

To find the factor, we pick a point P on the curve and attempt to compute:

$$Q = kP \pmod{N}$$

where k is a massive integer (the product of all primes up to some bound B). If g is smooth and g divides k, then in the shadow world modulo p, the result should be the identity (the Point at Infinity ($\mathcal{O}$)). How does this reveal p to us in the real world modulo N? Point addition on a curve involves calculating a slope λ. For two points (x_1, y_1) and (x_2, y_2), the slope is:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \pmod{N}$$

To compute this, we must find the **modular inverse** of $(x_2 - x_1) \pmod{N}$. We use the Extended Euclidean Algorithm to find inv such that:

$$inv \cdot (x_2 - x_1) \equiv 1 \pmod{N}.$$

If $\gcd(x_2 - x_1, N) = p$, the modular inverse **does not exist**. The Euclidean Algorithm will fail to find an inverse and will instead hand you the factor p. Thus, ECM is effectively a search for a curve where the point addition crashes because it hits a multiple of p.

YAFU launches thousands of curves in parallel.

- **Small a, b:** It starts with simple parameters.
- **Large Primes:** It uses a Two-Stage process. Stage 1 looks for points that are totally smooth. Stage 2 looks for points that are smooth except for one large prime factor, significantly increasing the success rate.

Magic #6: The SIQS and The Industrial Revolution of Factorization

When you are faced with a hard number—a product of two similarly sized primes, like an RSA key—Pollard's ρ and ECM will eventually hit a wall. To break through, we need the **Self-Initializing Quadratic Sieve (SIQS)**. This is not a random walk; it is an assembly line designed to build a solution.

The objective is based on **Fermat's Factorization Method**. If we can find integers x and y such that:

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

Then N must divide $x^2 - y^2 = (x - y)(x + y)$. Unless $x \equiv \pm y \pmod{N}$, calculating $\gcd(x - y, N)$ will yield a non-trivial factor. Finding such a pair by accident is impossible. SIQS **builds** them by combining smooth numbers.

A **quadratic residue** $a \pmod{p}$ is a number that has a square root in the field $\mathbb{F}_p$. In the Sieve, we only care about primes p where N is a quadratic residue. We collect these primes into our **Factor Base**.

Why? Because we are going to look for values of a such that $Q(a) = a^2 - N$ is smooth over this factor base. If N is not a residue mod p, then p can *never* divide $a^2 - N$, making it useless for our construction.

A sieve is the heart of the Magic. Instead of performing trial division on millions of $Q(a)$ values, we use a **Sieve Array**.

1. **Initialize:** Create an array where each index i represents a value a_i. Store the approximate logarithm $\log(Q(a_i))$ in each cell.
2. **Sieve:** For each prime p in the factor base, solve the congruence $a_i^2 \equiv N \pmod{p}$ to find the roots r_1, r_2.
3. **Jump:** Starting at r_1 and r_2, jump through the array in steps of p, subtracting $\log(p)$ from each cell you land on.
4. **Harvest:** After processing all primes, any cell that is **near zero** represents a $Q(a_i)$ that is almost certainly a product of the small primes in our factor base.

Once we have more smooth Relations than we have primes in our factor base, we represent each relation as a vector of exponents modulo 2. * A product of numbers is a perfect square if all its prime exponents are **even**. * We use **Gaussian Elimination** (or the Block Lanczos algorithm in YAFU) to find a subset of rows that sum to the zero vector modulo 2.

This sum of rows corresponds to a product of relations that forms our perfect square y^2. We take the square root, calculate the GCD, and the Dark Ages number is cracked.

Magic #7: The General Number Field Sieve (GNFS)

The Quadratic Sieve ($O(e^{\sqrt{\ln n \ln \ln n}})$) is limited because the numbers we are sieving ($a^2 - n$) grow too large. The **Number Field Sieve (NFS)** breaks this barrier by moving the problem out of the standard integers ($\mathbb{Z}$) and into **Algebraic Number Fields**.

In QS, we look for squares in one world: $\mathbb{Z}/n\mathbb{Z}$. In GNFS, we build two different mathematical worlds (rings) and a map between them: 1. **The Rational World:** The standard integers $\mathbb{Z}$. 2. **The Algebraic World:** A ring of algebraic integers $\mathbb{Z}[\alpha]$, defined by a polynomial $f(x)$ of degree d (usually 5 or 6).

We choose a polynomial $f(x)$ and an integer m such that $f(m) \equiv 0 \pmod{n}$. This creates a **homomorphism** ϕ that maps the algebraic world back to the integers mod n.

The goal is still to find $x^2 \equiv y^2 \pmod{n}$. In GNFS, we look for pairs (a, b) such that: * In the **Rational World**, the linear value $a - bm$ is smooth. * In the **Algebraic World**, the algebraic number $a - b\alpha$ has a smooth norm.

Because we are sieving over two different structures simultaneously, the numbers we need to check stay significantly smaller than $a^2 - n$ in the Quadratic Sieve. Smaller numbers are exponentially more likely to be smooth (the Dickman function works in our favor here!).

CADO-NFS is the specialized open-source suite used for these massive computations. Its magic lies in its orchestration: 1. **Polynomial Selection:** Finding the perfect $f(x)$ to keep the norms as small as possible. 2. **Sieving (The Monster):** This stage can run for months across thousands of CPUs. It uses a Lattice Sieve to find (a, b) pairs where both worlds are smooth. 3. **Filtering:** Throwing away redundant relations. 4. **The Matrix:** Solving a linear system with hundreds of millions of rows. 5. **Square Root:** Calculating the square root in the algebraic number field (a notoriously difficult task).

The complexity of GNFS is:

$$L_n[1/3, c] = e^{(c+o(1))(\ln n)^{1/3}(\ln \ln n)^{2/3}}$$

Notice the 1/3 exponent. This is the holy grail. It means that as n grows, the difficulty of factoring with GNFS increases much more slowly than with the Quadratic Sieve. This 1/3 vs. 1/2 shift is why we can factor 250-digit numbers today instead of waiting for the heat death of the universe.

YAFU: The Modern Orchestrator

YAFU (Yet Another Factorization Utility) is the gold standard general-purpose factorization routine. It is not just an implementation of these algorithms; it is a **highly adaptive expert system**. It analyzes the input and launches a multi-stage attack:

1. **Pre-Sieve:** It clears out the first 600,000 primes (up to 10M) using a fast bit-sieve.
2. **Rho-Stage:** It runs several walks of Pollard's ρ to catch the easy fruit. Run for 1 minute to find anything up to 15 digits.

3. **ECM-Stage:** It launches curvesets. If it finds a factor, it restarts the logic on the remaining composite part.
4. **SIQS-Stage:** If the number is still standing (usually when it's a balanced semiprime), YAFU initializes the Sieve.
5. **Bail to CADU-NFS:** If the number is still unfactored and over 120 digits, YAFU will generate a CADO-NFS compatible file and suggest you move to a cluster.

The reason this pipeline works is because of the **Dickman Function**. We start with algorithms that look for **Small Factors** (Rho, ECM) because small factors are statistically much more likely to exist. We only move to the **Global Sieve** (SIQS, GNFS) once we have a high degree of confidence that no small factors are left to find.

Transition from the Quadratic Sieve to the Number Field Sieve: Below 100 digits SIQS is faster. The overhead of choosing a polynomial and setting up the two-world map in GNFS is too high. SIQS's simpler one-world approach wins on sheer speed per iteration. Above 120 digits GNFS is the undisputed king. Because of the 1/3 exponent in its complexity, it scales much better. Even though each GNFS step is harder, you need far fewer relations to finish the job.

The magic you feel when using YAFU is the result of thousands of hours of optimization:

- **SIMD Parallelism:** The sieving code is written in assembly/intrinsics to use AVX-512 instructions, processing multiple array cells at once.
- **Self-Initialization:** In the SIQS, it changes the polynomial $Q(a)$ frequently to keep the values in the array small, maximizing the probability of smoothness (thanks to the Dickman function - less likely to be smooth further from $\sqrt{N}$). This is called the Multiple Polynomial Quadratic Sieve (MPQS).
- **The Matrix Solver:** It uses the Block Lanczos algorithm, which can solve a matrix with millions of rows in seconds using sparse matrix techniques.

Algorithm	Ideal Factor Size (p)	Input Size (N)	Complexity $O(f)$	The Hand-off Logic
Trial Division	1–6 digits	Any	$O(B)$	Always first. Clears the grass (tiny primes).
Pollard's ρ	7–15 digits	Any	$O(\sqrt{p})$	Effortless. If no factor appears in ~10M iterations, move on.
ECM	15–40 digits	Any	$O(e^{\sqrt{2\ln p \ln\ln p}})$	The Fisherman. Runs curves until the cost of a new curve > cost of SIQS.
SIQS	Balanced	50–110 digits	$O(e^{\sqrt{\ln N \ln\ln N}})$	The Factory. Best when N is a product of two similar primes.
GNFS	Balanced	110+ digits	$O(e^{\sqrt[3]{\ln N(\ln\ln N)^2}})$	The Super-Factory. Handed off to CADO-NFS for massive distributed runs.

Future Magic: The Quantum Leap and Shor's Algorithm

We've moved from the linear slog of the Dark Ages to the logarithmic elegance of modern number theory. We use the Birthday Problem to find collisions, Elliptic Curves to find smooth group orders, and high-dimensional linear algebra to weave perfect squares out of smooth straw.

The next time you see YAFU factor a 100-digit number in minutes, remember: it isn't just calculating—it's exploiting the deep, probabilistic architecture of the integers. But there's another revolution around the corner.

Just as we mastered the Industrial Age of factoring with the Quadratic Sieve, the horizon shifted. Enter **Shor's Algorithm** (1994). This is the Nuclear Option of number theory. While the Best Classical Algorithm (GNFS) is sub-exponential, Shor's is **polynomial time** ($O(\log^3 N)$).

Peter Shor realized that the problem of factoring an integer N can be reduced to the problem of finding the **period** (the order) of a function.

If you pick a random $a < N$, the function $f(x) = a^x \pmod N$ is periodic. Let r be the smallest integer such that:

$$a^r \equiv 1 \pmod N$$

If we can find this period r, and if r is even, we can write:

$$(a^{r/2})^2 - 1 \equiv 0 \pmod N$$

$$(a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \pmod N$$

And just like in our Quadratic Sieve, the factors of N are found by $\gcd(a^{r/2} \pm 1, N)$.

On a classical computer, finding the period r is just as hard as factoring itself (you'd have to calculate a^x until it repeats, which is the Dark Ages all over again). Shor's Algorithm uses two quantum superpowers to find r instantly:

1. **Quantum Superposition:** The computer creates a superposition of all possible values of x. It calculates $f(x) = a^x \pmod N$ for *all* x simultaneously in a single quantum state.
2. **Quantum Fourier Transform (QFT):** This is the magic. The QFT is used to measure the interference pattern of the periodic function. Just as a prism splits light into its frequencies, the QFT reveals the period r by collapsing the superposition into a state that represents the frequency of the function's repetition.

If Shor's is so fast ($O(\log^3 N)$), why isn't RSA dead? The catch is decoherence.

- **Scaling:** To factor an n-bit number, you need roughly $2n$ logical qubits. For a 2048-bit RSA key, that's 4096 perfect qubits.
- **Error Correction:** Because quantum states are fragile (decoherence), we need thousands of physical qubits to create one logical qubit. Current quantum computers (like Google's Sycamore or IBM's Osprey) are still in the hundreds of noisy physical qubits range.

Shor's algorithm proves that the security of modern encryption isn't based on a mathematical impossibility, but on a **hardware limitation**. We are currently in the NISQ (Noisy Intermediate-Scale Quantum) era. When Fault-Tolerant quantum computers arrive, the

Dark Ages won't just be over—the current age of RSA and Elliptic Curves will become history.

References

1. Crandall, R., & Pomerance, C. (2005). Prime Numbers: A Computational Perspective. Springer. (Crandall and Pomerance 2005) This is the absolute must-have. It covers everything we discussed: the Dickman function, the transition from Pollard's Rho to ECM, and the most detailed treatment of the Quadratic Sieve (and GNFS) available in print. It bridges the gap between pure number theory and the dirty implementation details used in YAFU.
2. Bach, E., & Shallit, J. (1996). Algorithmic Number Theory, Vol. 1: Efficient Algorithms. MIT Press. Why: This is the rigorous computer science view. It provides the complexity analysis for Magic #0 (GCD) and Magic #1 (Miller-Rabin). It's particularly good for understanding the Big-O limits of the Dark Ages.
3. Cohen, H. (1993). A Course in Computational Algebraic Number Theory. Springer. Why: Henri Cohen is a legend in the field (one of the creators of the PARI/GP system). This book is excellent for the Non-trivial roots of unity and the Elliptic Curve (ECM) sections. It is mathematically denser than Crandall-Pomerance but very rewarding for a PhD in mathematics.
4. Nielsen, M. A., & Chuang, I. L. (2010). Quantum Computation and Quantum Information. Cambridge University Press. Why: Known affectionately as "Mike and Ike," this is the definitive text for Shor's Algorithm. It explains the Quantum Fourier Transform (QFT) with the Prism clarity we discussed and provides the full proof of how period-finding reduces to factoring.

Postscript: Primes as Effectively Hyperuniform Systems

While this post treats primes as pseudo-random entities to be hunted and factored, there is a burgeoning field of research that views the primes through the lens of Statistical Mechanics and Scattering Theory.

There is hidden order in the prime forest. The paper *Uncovering Multiscale Order in the Prime Numbers via Scattering,* (Torquato et al. 2018) treats prime numbers along the number line as a collection of atoms. When you perform a scattering experiment on these atoms (essentially a Fourier transform of the prime locations), you don't see the chaotic noise of a random gas. Instead, you see a pattern called Effective Hyperuniformity.

Primes are not perfectly periodic like a crystal (which would make factoring trivial), but they are not disordered like a Poisson process. They exist in a state of constrained disorder. The scattering pattern of primes shows peaks (similar to how X-rays interact with crystals) at all rational frequencies. This suggests that the primes are aware of the sieve structure at every scale simultaneously. These are called Bragg-like Peaks. There is an apparent multiscale order: while local gaps between primes look random (the Dark Ages view), the global distribution of primes is incredibly rigid.

For the factorizer, this is a reminder that the Magic we use—the GCD, the Quadratic Sieve, the QFT—is only possible because we are exploiting a deep, multiscale harmony

that persists despite the apparent chaos of the number line.

References

Crandall, R. E., & Pomerance, C. (2005). *Prime numbers: a computational perspective*. Springer.

Torquato, S., Zhang, G., & De Courcy-Ireland, M. (2018). Uncovering multiscale order in the prime numbers via scattering. *Journal of Statistical Mechanics: Theory and Experiment*, *2018*(9). https://doi.org/10.1088/1742-5468/aad6be