Factorials and an Efficient Algorithm

Mahesh Ramani

September 2025

Introduction

Factorials are pretty cool. I think a lot of us felt a spark of interest back in high school when we learned that all those combinations and permutations could be handled with a single factorial operation. Even aside from its weird notation (an exclamation point, really?) the core idea seems almost trivial. It's just multiplying consecutive numbers, something we all did as kids playing with calculators.

It's no surprise that factorials are a huge deal. They're the foundation of combinatorics, and they pop up everywhere, like in Taylor series expansions. But they're also kind of a pain. Their growth is insane, shooting up to astronomical numbers almost instantly. You can see this easily on a graph:

Because they get so huge so fast, calculating $n!$ for a large n is tough. This is especially true for computers. They use a fixed number of bits to store integers, and even a 64-bit integer (the biggest standard type in most languages) can only hold values up to 20!. Trying to get around this with floating-point math just introduces a bunch of rounding errors, which kills your precision.

So, I was looking at factorials today and decided to try making a formula to calculate them. The one I came up with is pretty efficient—it has a bit complexity of

$$\mathcal{O}(n^{3/2} \log^2 n).$$

For practical purposes (think n up to around 5 billion), it works well. That makes it about $\mathcal{O}(\frac{\sqrt{n}}{\log n})$ times faster than the naive multiplication method. So, let's walk through how it works.

1 Core idea: prime factorization

Instead of multiplying $1 \cdot 2 \cdot 3 \cdots n$, let's take advantage of the fact that every number can be expressed in terms of its prime factorization (we used a similar trick when we looked at Carmichael numbers). Since $n!$ multiplies every number from 1 to n, we know that all primes in its factorization will be less than or equal to n.

Luckily, Legendre's formula gives us an easy way to calculate the exponent of each prime in the factorization of $n!$. The version of the formula we use in the code is:

$$V_p(n!) = \frac{n - S_p(n)}{p - 1},$$

where $S_p(n)$ is the sum of the digits of n when written in base p.

The idea is straightforward from there: we use a sieve to get all primes up to n, apply Legendre's formula to each one, and then multiply the results. But that's not fully optimized. Let's use some more math to make it easier.

A more practical form

There's another, more common form of Legendre's formula that's easier to work with:

$$E_p(n!) = \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \left\lfloor \frac{n}{p^3} \right\rfloor + \cdots$$

You get the idea—when p^k outgrows n, the summation dies, and the sum of the terms that did give us an integer results in the number of times p can divide $n!$. This works for all $0 < p \leq n$, but it can be simplified if we focus on certain values of p.

Say $p > \sqrt{n}$. Then $p^2 > n$. There we go—now the number of times p can divide into $n!$ can be simply represented as $\lfloor n/p \rfloor$. Taking it even further, we can solve for p in terms of n in $\lfloor n/p \rfloor$ by setting the expression equal to arbitrary exponents. Setting it equal to 1 gets us:

$$\lfloor n/p \rfloor = 1.$$

It follows that

$$1 \leq \frac{n}{p} < 2,$$

which splits into

$$\frac{n}{2} < p \leq n.$$

The pattern generalizes according to the inequality

$$\frac{n}{k+1} < p \leq \frac{n}{k},$$

where k is the p-adic valuation (essentially a kind of "floored" logarithm base p of n) of n. Remember, this pattern only stays for $p > \sqrt{n}$. However—fret not: $n - \sqrt{n} > \sqrt{n}$ for large n, so we cover the majority of cases using this pattern anyway. We can just use the original form of Legendre's formula for $p \leq \sqrt{n}$.

This pattern is a huge win for our algorithm. Instead of processing each prime individually, we can group all primes with the same value of k together, find their product, and then raise that product to the k-th power. This dramatically reduces the number of multiplications we need to do.

2 Sieving efficiently

Now, let's talk about how we find those primes. We're using the Sieve of Eratosthenes, just like we did with the Carmichael numbers. The goal is to avoid allocating a huge array of size n.

The solution is segmented sieving.

1. First, we sieve a small list of "base primes" up to $\sqrt{n}$.
2. Then, we break the range $(\sqrt{n}, n]$ into manageable chunks.
3. For each chunk, we mark off multiples of the base primes. This works because every composite number $m \leq n$ must have at least one prime factor $p \leq \sqrt{m} \leq \sqrt{n}$.

This method lets us sieve primes up to a million (and beyond!) without storing enormous arrays in memory.

3 Putting it all together

When we actually construct the factorial, we use a balanced product tree to combine the numbers efficiently, leveraging a fast multiplication library like `gmpy2`.

Obviously, for a large n, we can't just print the entire integer value of $n!$—it would be enormous. So instead, we build an approximation using scientific notation. Since $\log(n!) \approx \log(a \cdot 10^b)$, we can use the logarithm of our final product to find a and b.

And since we're introducing floating-point arithmetic with these logarithms, I also added a percent error calculator that uses Stirling's approximation to check our work.

Wrapping up

And that's the core of it! This walkthrough mainly focused on the underlying math—the tricks with prime factorization and Legendre's formula that make calculating huge factorials efficient. I've skipped over some of the nitty-gritty implementation details, so if you're curious to see how all these pieces fit together in practice, feel free to check out the actual code to see it all work out.