Calcuating n! Efficiently

Mahesh Ramani
mahesh.ramani.iyer@gmail.com

September 25, 2025

Abstract

This note presents a summary of an algorithm for computing $n!$ based on prime factorization, Legendre's formula, segmented sieving, and grouped multiplications (implemented with a balanced product tree and fast big-integer arithmetic). The method trades the straightforward repeated multiplication of $1 \cdot 2 \cdot \ldots \cdot n$ for a decomposition into prime powers, groups primes with identical p-adic valuations of n when sensible, and uses efficient multiplication primitives (e.g. `gmpy2`). The practical outcome is substantial savings in the number of multiplications and memory when the goal is either an exact result for moderate n or a high precision scientific notation approximation for very large n.

1 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. Naively computing

$$n! = \prod_{k=1}^{n} k$$

by multiplying the integers one after another becomes impractical quickly: the integer size grows super-exponentially in the input size (the bit-length of $n!$ is $\Theta(n \log n)$), and naive multiplication repeats many costly large-integer multiplications. Moreover, fixed-width numeric types overflow at small n (e.g. 20! exceeds 64-bit integer limits), so arbitrary-precision arithmetic and better algorithms are required for large n.

Some common approaches of doing this include:

- *Schoolbook sequential multiplication:* iterate multiplication $1 \cdot 2 \cdot \ldots \cdot n$; simple but wasteful in both work and intermediate memory.
- *Divide-and-conquer product trees:* multiply pairs in a balanced manner to lower the depth of multiplications and exploit fast multiplication algorithms.
- *Prime-factor based methods:* express $n!$ as $\prod_{p \le n} p^{e_p}$ where each exponent e_p is given by Legendre's formula and then form the product.

We follow the last approach (prime factorization) and dig into two other practical optimizations that appear in the provided implementation:

1. segmented sieving to enumerate primes up to n without allocating an $O(n)$-sized array in memory, and
2. grouping primes with identical values of the quotient $\lfloor n/p \rfloor$ (for large primes) and using product trees plus exponentiation to reduce repeated work.

All main notation and the key lemma used in the method are summarized next.

2 Notation and key facts

- n — the input integer whose factorial is required.
- p — a prime number.
- $\nu_p(m)$ — the exponent of prime p in the prime-factorization of integer m.
- $e_p(n) \equiv \nu_p(n!)$ — exponent of p in $n!$.
- $\lfloor x \rfloor$ — floor of real x.
- log without subscript is the natural logarithm; $\log_{10}$ is base-10 log.
- mpz, mpfr — big-integer and precise floating types (used by `gmpy2` in the implementation).

[Legendre] For any prime $p \leq n$,

$$e_p(n) = \sum_{k \geq 1} \left\lfloor \frac{n}{p^k} \right\rfloor,$$

where the sum is finite because $\lfloor n/p^k \rfloor = 0$ for $p^k > n$.

A useful reformulation for the implementation: for primes $p > \sqrt{n}$ the sum reduces to the single term $\lfloor n/p \rfloor$, because $p^2 > n$.

3 Algorithm walkthrough

Below is a clean, step-by-step description of the algorithm in n!.py.

Inputs and outputs

Input: integer $n \geq 0$. Optional parameters control whether the full integer $n!$ is returned (or only a scientific-notation approximation) and the number of significant digits of the approximation.

Output: either

- the exact $n!$ as a multi-precision integer (if feasible given memory limits), or
- a precise scientific-notation approximation of $n!$, together with metadata (estimated number of digits, number of primes used, time taken).

Steps

1. **Enumerate primes up to n.** Use a segmented Sieve of Eratosthenes to compute the list $\mathcal{P} = \{p \le n\}$. The implementation sieves base primes up to $\lfloor\sqrt{n}\rfloor$ with a standard in-memory sieve, then processes the interval $(\sqrt{n}, n]$ in chunks to avoid allocating an $O(n)$ array. This is implemented as `segmented_sieve` in the code.

2. **Split primes at $\sqrt{n}$.** Let

$$\mathcal{P}_{\text{small}} = \{p \in \mathcal{P} : p \le \sqrt{n}\}, \qquad \mathcal{P}_{\text{large}} = \mathcal{P} \setminus \mathcal{P}_{\text{small}}.$$

   For each $p \in \mathcal{P}_{\text{small}}$ compute $e_p(n)$ via Legendre's formula (implemented as `legendre_exponent`). This is used for p in the range $(\sqrt{n}, n]$. For $p > \sqrt{n}$,

$$e_p(n) = \left\lfloor \frac{n}{p} \right\rfloor.$$

   Setting $k := \lfloor n/p \rfloor$ yields the interval

$$\frac{n}{k+1} < p \le \frac{n}{k}.$$

   Thus primes with the same k lie in the same interval and share exponent k in $n!$. In particular:

   - $k = 1$ corresponds to primes in $\left(\frac{n}{2}, n\right]$ (each contributes exponent 1),
   - $k = 2$ corresponds to primes in $\left(\frac{n}{3}, \frac{n}{2}\right]$ (each contributes exponent 2),
   - and so on.

   Because we only consider primes $p > \sqrt{n}$ here, we have $k = \lfloor n/p \rfloor < \lfloor n/\sqrt{n} \rfloor = \lfloor \sqrt{n} \rfloor$. Therefore the number of nonempty groups indexed by k is at most $\lfloor\sqrt{n}\rfloor$. To oversimplify a bit, we use the inequalites we just derived to throw p's with the same k into a group. We take the group product and then raise it to the kth power, significantly reducing the amount of multiplication we do.

3. **Group large primes by quotient.** For large primes it is efficient to group primes by the value $k = \lfloor n/p \rfloor$. For a fixed $k \ge 1$, all primes in that group contribute the same multiplicative exponent k in $n!$. Compute the product $G_k = \prod_{p \in \text{group}_k} p$ (using a product tree) and then include G_k^k in the global product. In the code this grouping uses a `defaultdict(list)` keyed by $k = \lfloor n/p \rfloor$ and computes each group product via `product_tree_iter`. This decreases repeated multiplications compared to multiplying each prime k times individually.

4. **Assemble the factorial.**

   - If the estimated decimal digit-length of $n!$ is small enough (controlled by a threshold in the implementation), construct the full integer by exponentiating each group product and small-prime base using fast integer power (e.g. `gmpy2.pow`) and combining them using a balanced product tree.
   - Otherwise only compute $\log_{10}(n!)$ by accumulating contributions $e_p(n) \log_{10} p$ and format a scientific-notation approximation with the requested number of significant digits. The implementation produces both an estimated digit count and a leading-digit string for display.

4 Complexity analysis

Below we analyze both time and space complexity. Where more precise bit-cost estimates are needed we point out the required assumptions (especially on the cost of large-integer multiplication).

4.1 Sieve (enumerating primes up to n)

A classical Sieve of Eratosthenes up to n runs in time

$$T_{\text{sieve}}(n) = O(n \log \log n)$$

arithmetic operations (this is the usual asymptotic bound for the sieve when implemented carefully). The segmented sieve used in the implementation achieves the same time bound while keeping peak memory to $O(\sqrt{n} + \text{chunk\_size})$. Practically the code uses base primes up to $\sqrt{n}$ and then processes blocks of size at least $\sqrt{n}$ or a tuned constant (see `segmented_sieve` implementation). Thus:

$$\text{Time} = O(n \log \log n), \qquad \text{Space} = O(\sqrt{n}) \text{ (for sieve buffers / base primes).}$$

4.2 Computing exponents for small primes

For each $p \le \sqrt{n}$ the code computes

$$e_p(n) = \sum_{k \ge 1} \left\lfloor \frac{n}{p^k} \right\rfloor$$

by iterating powers $p, p^2, p^3, \ldots$ until $p^k > n$. The number of iterations for a given p is $O(\log_p n)$. Summing over all small primes yields

$$\sum_{p \le \sqrt{n}} O(\log_p n) = O\Big(\sum_{p \le \sqrt{n}} \frac{\log n}{\log p} \Big).$$

Using crude bounds ($\pi(x) \approx x/\log x$) this sum is much smaller than n and typically dominated by the sieving cost for practical ranges of n. Consequently it is not the dominant cost in practice.

4.3 Grouped large primes and product formation

Let $\mathcal{P}_{\text{large}}$ be the primes in $(\sqrt{n}, n]$. There are about $\pi(n) - \pi(\sqrt{n}) \approx n/\log n$ such primes. Grouping them by $k = \lfloor n/p \rfloor$ reduces repeated exponentiations: for group k we compute $G_k = \prod_{p \in \text{group}_k} p$ once and then form G_k^k.

The asymptotic cost of forming these group products depends heavily on the cost model for multiplying large integers. Denote by $M(B)$ the cost (in bit operations) to multiply two B-bit integers. Modern libraries (e.g. GMP which `gmpy2` wraps) use subquadratic methods (Karatsuba, Toom–Cook, FFT-based) so $M(B)$ is roughly $O(B \log B)$ or better for very large B.

If the algorithm constructs the full $n!$, the final integer has bit-length $B = \Theta(n \log n)$. Combining product-tree techniques with fast multiplication leads to a total bit-cost which is asymptotically on the order of $O(M(B) \log B)$ in many standard analyses of product-tree construction; hence the multiplication stage typically dominates the running time for large n.

In short:

- If we only need an approximation (logarithm of $n!$), the dominant cost is sieving: time $O(n \log \log n)$ and space $O(\sqrt{n})$.

- If we need the exact integer $n!$, the dominant cost is large-integer multiplication. The bit-complexity is approximately that of performing a sequence of $O(\log n)$ product-tree multiplications on numbers of size up to $B = \Theta(n \log n)$ bits, which yields an overall bit-cost in terms of $M(B)$ (multiplication cost of B-bit numbers). Using a fast multiplication model $M(B) = O(B \log B)$ yields a total bit-cost around $O\big(B \log B \cdot \log B\big)$ in a naive bound; practical performance is much better and benefits from grouping and balanced product trees.

4.4 Memory (space) considerations

- **Sieve buffers and base primes:** $O(\sqrt{n})$ memory.
- **Storing primes:** storing $\pi(n) \approx n/\log n$ primes requires $O(n/\log n)$ words (but each prime fits in $O(\log n)$ bits). The segmented sieve allows streaming primes and grouping them without holding an $O(n)$ Boolean array.
- **Full integer build:** if the full $n!$ is built, the memory needed to hold the final integer is $O(n \log n)$ bits. The implementation protects against building giant integers by estimating digit count and only building the full integer when it is feasible (controlled by the `max_full_digits` parameter). See the code's metadata output for the `built_full` flag.

5 Implementation notes

- **Language:** Python (reference implementation `n!.py`).
- **Libraries:** `gmpy2` (fast multi-precision arithmetic), Python `math`, and `collections.defaultdict`.
- **Key choices:** segmented sieve with tunable chunk size (default $\max(\sqrt{n}, 32768)$); product-tree implementation `product_tree_iter`; grouping by $\lfloor n/p \rfloor$; heuristic threshold for building the full integer (`max_full_digits`).

6 Empirical and theoretical discussion

6.1 Insights from the approach

- **Factor once, exponentiate many:** grouping primes into G_k and computing G_k^k avoids repeated multiplications of the same prime and concentrates work into fewer, larger multiplications.
- **Segmentation for memory:** streaming primes keeps peak memory low and enables handling much larger n than a naive full sieve.
- **Balanced product trees:** reduce multiplication depth and allow the routine to exploit fast multiplication asymptotically.

6.2 Trade-offs and limitations

- The algorithm's asymptotic bottleneck when producing the exact $n!$ is large-integer multiplication; asymptotically, multiplication cost grows with the bit-size of $n!$, that is $\Theta(n \log n)$ bits. Thus, while the algorithm reduces the number of operations compared to naive multiplication, the cost of handling very large integers remains intrinsic.

- The grouping strategy helps most when there are many primes that share small quotients k. For certain ranges or special n the grouping advantage might be less pronounced.
- The implementation makes pragmatic choices (chunk sizes, threshold for building the full integer) that are tuned for typical hardware; extreme inputs may need retuning.

6.3 Does theory match practice?

Yes, qualitatively:

- For producing scientific-notation approximations and estimating digit counts, the algorithm is essentially dominated by prime enumeration; time and memory are controlled and match the $O(n \log \log n)$ sieve behavior in practice.
- For producing the exact integer, the practical runtime is dominated by the big-integer multiplications; grouping and product trees noticeably reduce wall-clock time compared to naive sequential multiplication, and the code benefits from optimized GMP routines via `gmpy2`. Empirical measurements (timings printed in the reference script) confirm the expected speedups for moderate values of n. See the script's main invocation and its printed metadata for example runs and timing outputs.

7 Conclusion

We compute $n!$ by enumerating primes with a segmented sieve, obtaining prime exponents via Legendre's formula, grouping large primes by $k = \lfloor n/p \rfloor$, and assembling the result using balanced product trees and fast big-integer routines. For primes $p > \sqrt{n}$ this grouping lets us compute each group product once and raise it to the appropriate power, avoiding many redundant multiplications. The key takeaway is that prime-factor decomposition combined with grouping and balanced multiplication substantially reduces work and memory compared to naive sequential multiplication, making both exact computation and high-precision approximation of $n!$ far more practical for large n.

A Representative pseudocode

Listing 1: Pseudocode (closely follows n!.py)

```
# 1. primes = segmented_sieve(n)
# 2. split primes into small (<= sqrt(n)) and large (> sqrt(n))
# 3. small_factors = [(p, legendre_exponent(n,p)) for p in small_primes]
# 4. group large_primes by k = n // p, for each group compute G_k =
   product_tree(group)
#    then include (G_k ** k) as a factor
# 5. if estimated_digits <= threshold:
#       compute exact product via product_tree over all factors (fast mpz
   arithmetic)
#    else:
#       compute log10(n!) by summing e_p*log10(p) and format scientific
   notation
```