Exact Computation of the Catalan Number $C(2{,}050{,}572{,}903)$

Mahesh Ramani
mahesh.ramani.iyer@gmail.com

November 2025

Abstract

I describe a two-phase algorithm for computing exact Catalan numbers at unprecedented scale. My method computes the Catalan number

$$C(n) = \frac{(2n)!}{n!\,(n+1)!},$$

for $n = 2,050,572,903$ (a result with a targeted $1,234,567,890$ decimal digits) by operating on prime factorization data rather than forming giant factorials. This happens in a two-phase process: Phase 1 uses parallel segmented sieve to sieve primes up to $2n$, and Legendre's formula is used to compute each prime's multiplicity in the prime factorization of $C(n)$. I group primes by exponent and write the prime factorization to disk. Phase 2 uses a product tree to reconstruct the integer from the factorization, using chunking to limit memory. To the best of my knowledge, this computation produces the largest exact Catalan number ever computed. I report performance statistics (wall times, memory) on a single-machine run, and provide implementation details (scripts `catalan.py`, `catalanreconstructor.py`) to ensure reproducibility. The final result's SHA-256 hash is recorded for verification. Verification strategies include independent reconstruction from the factor file and modular checks on low-order bits.

1 Introduction

Catalan numbers appear in many combinatorial problems (e.g. binary trees, polygon triangulations) and grow exponentially with n, given their asymptotic bound of 4^n times a polynomially decaying factor. By definition,

$$C(n) = \frac{(2n)!}{n!\,(n+1)!},$$

cannot be computed directly for large n due to the astronomical size of $(2n)!$. Traditional recursive or dynamic programming methods become infeasible for n on the order of millions, let alone billions. Instead, I use a prime-factorization approach: by Legendre's formula, one

can compute the exponent of each prime $p \leq 2n$ in the numerator and denominator, thereby determining the exact prime-power factorization of $C(n)$. Grouping primes by exponent and reconstructing via balanced product trees yields the final integer.

I implement this approach in two phases. Phase 1 (in `catalan.py`) sieves primes up to $2n$ using a parallel segmented sieve and computes each prime's exponent in the factorization of $C(n)$. The primes are grouped by exponent e and written to a factorization file `catalan_[n]_factorization.txt`, where [n] represents the value of n. In Phase 2 (in `catalanreconstructor.py`), the file is read and processed group-by-group: for each exponent e, I compute

$$G_e = \prod_{p \in \text{group}_e} p$$

and then compute $P_e = G_e^e$. The final Catalan number is $\prod_e P_e$, giving us:

$$C_n = \prod_e \left(\left(\prod_{p \in \text{group}_e} p \right)^e \right)$$

where the inner and outer products are computed using memory-efficient balanced product trees.

All large multiprecision arithmetic is done with `gmpy2` (GMP). Intermediate results are written to disk (text and binary files) so that Phase 1 and Phase 2 can be run on separate machines if needed.

This paper presents the mathematical foundations (Section 2), algorithmic details (Section 3), and implementation (Section 4), along with pseudocode for each phase. In Section 5 I report experimental results and resource usage for the computation $n = 2,050,572,903$. Section 6 discusses bottlenecks, future improvements, and time/space complexities. I conclude by describing verification strategies, including the provided SHA-256 hash of the final result, to enable independent validation. All code, data, and output files for this computation are available on my GitHub[1].

2 Mathematical Foundation

Let $v_p(m)$ denote the exponent of prime p in the prime-factorization of integer m (the p-adic valuation). Legendre's formula states that for positive integers m and prime p,

$$v_p(m!) \;=\; \sum_{k=1}^{\infty} \left\lfloor \frac{m}{p^k} \right\rfloor,$$

a finite sum since $\lfloor m/p^k \rfloor = 0$ for $p^k > m$. Using this, I can express the p-adic valuation of the Catalan number:

$$v_p(C(n)) \;=\; v_p((2n)!) \;-\; 2\,v_p(n!) \;-\; v_p(n+1).$$

[1]Reconstructed and output files are in README+OUTPUTFILES.md.

Thus each prime $p \leq 2n$ contributes an exponent

$$e_p = v_p((2n)!) - 2v_p(n!) - v_p(n+1),$$

and $C(n) = \prod_{p \leq 2n} p^{e_p}$. Note $v_p(n+1)$ is nonzero only for primes dividing $n+1$. Because $C(n)$ is known to be an integer (e.g. by combinatorial interpretation or Kummer's theorem), all e_p computed as above are nonnegative. Working with these exponents avoids ever forming the enormous factorials $(2n)!$, $n!$, etc.

3 Algorithm Overview

My computation proceeds in two phases. First I generate the factorization of $C(n)$ (in prime-exponent form), and then I reconstruct the integer from that factorization.

3.1 Phase 1: Factorization Generation (`catalan.py`)

I enumerate all primes $p \leq 2n$ using a parallel segmented Sieve of Eratosthenes. The sieve is implemented in Python, first finding all primes $\leq \sqrt{2n}$ and then sieving disjoint intervals in parallel (using `multiprocessing.Pool`). For each prime p found, I compute

$$a_p = v_p((2n)!), \quad b_p = v_p(n!), \quad c_p = v_p(n+1),$$

using Legendre's formula. Then $e_p = a_p - 2b_p - c_p$. For primes $n < p \leq 2n$, the term $v_p((2n)!) - 2v_p(n!)$ simplifies to either 1 or 0 (since $\lfloor 2n/p \rfloor - 2\lfloor n/p \rfloor$ is 1 iff p appears once in the binomial $\binom{2n}{n}$); the code handles this case automatically. Next, the script factors $n+1$ by trial division using the prime list and subtracts those valuations c_p from each e_p.

I then group the primes by their exponent e_p. Let G_e denote the set of all primes with exponent e. In my implementation, I write the factorization to the text file `catalan_2050572903_factorization.txt` in the format:

```
# Prime factorization of Catalan(n)
# exponent=E count=K
p1 p2 p3 ...   (primes with exponent E)
...
```

Each block lists all primes with the same exponent E. In parallel, the code accumulates a high-precision approximation of $\log_{10} C(n)$ to estimate the digit count (which is then output as metadata). The pseudocode summarizing this phase is found in Algorithm 1.

3.2 Phase 2: Reconstruction (`catalanreconstructor.py`)

Here, we read the factorization file and reconstruct $C(n)$. The file is parsed group by group. For each exponent e with prime group G_e, I compute the product

$$Q_e = \prod_{p \in G_e} p.$$

Algorithm 1 Factorization Generation for Catalan(n) (in `catalan.py`)

Require: Integer n.

Compute all primes $\leq 2n$ via parallel segmented sieve.
Factor $m = n + 1$ by trial division with primes $\leq \sqrt{n+1}$.
for each prime $p \leq 2n$ **do**
  Compute $a = \sum_{k \geq 1} \lfloor 2n/p^k \rfloor$.
  Compute $b = \sum_{k \geq 1} \lfloor n/p^k \rfloor$.
  Set $e = a - 2b$.
  if p divides $n + 1$ with exponent c **then**
    Set $e = e - c$.
  end if
  if $e > 0$ **then**
    Add p to group G_e.
  end if
end for
Write each group G_e to `catalan_`n`_factorization.txt`.
Estimate and output digit count of $C(n)$.

Algorithm 2 Integer Reconstruction for Catalan(n) (in `catalanreconstructor.py`)

Require: Factorization groups G_e from file.

Initialize list $\mathcal{P} = []$.
for each exponent e with prime list G_e **do**
  Compute $Q_e = \text{BalancedProductTree}(G_e)$.
  Compute $P_e = Q_e^e$ using fast pow.
  Append P_e to $\mathcal{P}$.
end for
Compute $C(n) = \text{BalancedProductTree}(\mathcal{P})$.
Write $C(n)$ to `catalan_`n`_reconstructed.bin`.

This is done using a memory-efficient balanced product tree once again: I multiply the list of primes by repeatedly pairing and multiplying subproducts (and periodically garbage-collecting) to limit intermediate sizes. Once Q_e is obtained, I compute $P_e = Q_e^e$ (via `gmpy2`). Finally, the Catalan number is $\prod_e P_e$. This final product is again computed by a balanced product tree over all P_e values. The result is written to `catalan_2050572903_reconstructed.bin` (binary format) and optionally to a newline-formatted decimal text for inspection.

Here, in Algorithm 2, BalancedProductTree(S) denotes a recursive divide-and-conquer multiplication of elements in set S. In practice, the primes in a large group G_e or the list $\mathcal{P}$ may be huge, so the code multiplies in chunks (e.g. 100,000 primes at a time) and combines partial products, with periodic garbage collection and a hybrid strategy combining disk and memory-based multiplication to constrain RAM usage.

4 Implementation and Reproducibility

The code is implemented in Python 3 using standard libraries. Key files are:

- `catalan.py`: Phase 1 factorization generation.
- `catalanreconstructor.py`: Phase 2 integer reconstruction.

The latter uses the `gmpy2` library (which wraps GMP) for multiprecision arithmetic. The former uses `multiprocessing` for parallel sieve segments.

To reproduce my result, one should run `catalan.py` with $n = 2,050,572,903$, which will produce `catalan_2050572903_factorization.txt`. Then run `catalanreconstructor.py` on that factorization file to produce `catalan_2050572903_reconstructed.bin`. I provide these output files for reference. All scripts include parameters (segment size, chunk size) with default values that worked for this run; these can be tuned for other hardware. More information on my computer specifications is found in Section 5.

5 Results

I ran the two phases sequentially on a single machine with the above specifications. The key statistics of the run are summarized in Table 1. Notably, the input $n = 2\,050\,572\,903$ yielded a Catalan number with exactly 1 234 567 890 decimal digits (as targeted). The final binary result file has size 512,643,220 bytes (= 512.643 MB decimal = 488.895 MiB) and SHA-256 checksum `dac68f4ee35db8e9400e68bd6140e6cbccec6fb8ce81059318400e2c44e45ae4`. The total wall-clock time was 1549.14 seconds, with 179.37 s in Phase 1 and 1369.77 s in Phase 2 (measured with `time`). Peak memory usage was about 4 GB during Phase 2; Phase 1 used much less (not tightly measured).

Table 1: Computation statistics for $C(2,050,572,903)$.

Statistic	Value
n	2,050,572,903
Decimal digits of $C(n)$	1,234,567,890
Bit length of $C(n)$	4,101,145,759
Size of result file	488.9 MiB
SHA-256 of output	found above
Phase 1 runtime (wall)	179.37 s
Phase 2 runtime (wall)	1369.77 s
Total wall time	1549.13 s
Peak memory (Phase 2)	4 GB
Hardware	AMD Ryzen 9 4900HS, 16 GB RAM, 6 GB GPU (unused),
OS	Windows
Software	Python 3, gmpy2

These results demonstrate the feasibility of computing Catalan numbers in the billion-digit range with commodity hardware. In particular, I believe (to the best of my knowledge)

that $C(2\,050\,572\,903)$ is the largest exact Catalan number computed so far. I provide the factorization file and final output hash so that others can independently verify the result.

6 Discussion

The two-phase pipeline—factorization followed by reconstruction—is central to the algorithm's feasibility at this scale. Phase 1, the factorization, can be executed on machines with modest RAM, as its primary tasks are sieving for primes and computing Legendre's formula exponents. The dominant bottleneck, both in terms of computational time and memory, is the reconstruction phase. The large-integer multiplications required to combine the prime powers consume the majority of the runtime and dictate the hardware requirements for extreme-scale computations.

Disk I/O can also become a limiting factor during reconstruction, particularly when intermediate products are offloaded to disk to manage memory. For this reason, high-throughput storage such as NVMe or other solid-state drives is strongly preferred over traditional spinning disks to mitigate this potential bottleneck. Ultimately, the performance of Phase 2 is bound by the available RAM and the efficiency of the underlying multiprecision arithmetic library.

6.1 Time Complexity

- **Sieve:** Enumerating primes up to $2n$ costs approximately $\mathcal{O}(n \log \log n)$ work. The sieve benefits significantly from parallelization.
- **Exponent Computation:** For each prime p, the computation of its exponent via Legendre's formula involves a sum of floor divisions. The cost of this step is subdominant compared to the reconstruction phase for the large values of n targeted by this work.
- **Reconstruction:** This phase is dominated by large-integer multiplications. Let B be the bit-length of the final integer, where $B = \Theta(n \log n)$. If $M(B)$ denotes the cost of multiplying two B-bit integers, the reconstruction cost is approximately $\mathcal{O}(M(B) \log P)$, where P is the number of distinct prime factors. In practice, using GMP's advanced multiplication algorithms, this behaves asymptotically like $\Theta(n(\log n)^2)$ bit-operations.

6.2 Space Complexity

The peak memory usage is dictated by the largest intermediate integer held in memory during the balanced product-tree reductions in Phase 2. The size of this integer, and thus the space complexity, is determined by the bit-length of the final Catalan number, which is $\Theta(n \log n)$ bits. The storage for the prime numbers generated during the sieve requires $\mathcal{O}(n/\log n)$ machine words. The two-phase design effectively trades increased disk I/O for a reduction in peak RAM requirements by storing the complete factorization on disk before reconstruction begins.

7 Conclusion

I have presented a practical and reproducible method for computing exact Catalan numbers at a scale previously unreported. The successful computation of $C(2,050,572,903)$ demonstrates the effectiveness of the prime factorization approach. The decoupling of factorization and reconstruction allows for a flexible use of hardware, where the more demanding reconstruction phase can be performed on a machine equipped with sufficient memory and processing power. The primary bottlenecks remain the large-integer multiplications and potential I/O limitations during reconstruction. I have provided the implementation scripts and the resulting factorization data to encourage independent verification and further research into optimizing and scaling these methods for even larger values of n.

The magnitude of the number I computed is chiefly a testament to the algorithm that produced it rather than to the number itself. Nevertheless, Catalan numbers have many combinatorial interpretations; one classical interpretation gives a neat closing problem.

Problem. How many ways can an 2,050,572,905-sided polygon be triangulated into 2,050,572,903 triangles?

Solution. The exact reconstructed integer is provided in the accompanying output file `catalan_2050572903_reconstructed.bin`.