arXiv:2601.11621v1 [cs.DS] 11 Jan 2026

卡特兰数的精确计算C(2，050，572，903)

Mahesh Ra
mani独立研究者

2025年11月

摘要

本文提出了一种两阶段算法，可在前所未有的规模下精确计算卡特兰数。文中通过计算 $C(n)$ 进行演示：当 n =2，050，572，903 时，所得结果的目标长度为 1，234，567，890 位十进制数。为规避计算大型阶乘所带来的内存限制，该算法完全在素数指数域中运行。阶段 1 使用并行分段筛枚举不超过 $2n$ 的素数，并应用勒让德公式确定 $C(n)$ 的精确素因数分解。素数按指数分组并序列化到磁盘。阶段 2 使用支持分块的内存高效平衡乘积树重构最终整数。该算法的时间复杂度为 $\Theta(n(\log n)^2)$ 位运算，空间复杂度为 $\Theta(n\log\ n)$ 位。这是迄今为止计算出的最大精确卡特兰数。文中报告了单机运行的性能统计数据，并讨论了包括模校验和 SHA-256 哈希验证在内的验证策略。文中还提供了源代码和因式分解数据，以确保结果可复现。

关键词：卡特兰数、阶乘、任意精度算术、组合数学。

1 引言

卡特兰数出现在许多组合数学问题中（例如二叉树、多边形三角剖分） [4] ，并且随着 n 指数级增长，其渐近增长为 $C(n) \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$。根据定义，

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

由于 $(2n)!$ 的规模极其庞大，无法直接计算较大的 n 。对于数量级达到数百万、更不用说数十亿的 n ，传统的递归或动态规划方法已不再可行。另一种方法是采用素因数分解方法：利用勒让德公式[1], ，可以计算分子和分母中每个素数 $p \le 2n$ 的指数，从而确定 $C(n)$ 的精确素数幂分解。按指数对素数分组，再通过平衡乘积树重构，即可得到最终整数。

此方法分两个阶段实现。阶段 1 使用并行分段筛筛出不超过 $2n$ 的素数，并计算每个素数在 $C(n)$ 的因式分解中的指数。这些素数

按指数分组，并写入因式分解文件。在阶段 2 中，读取该文件并逐组处理：对于每个指数 e,

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

并计算出 $P_e = Q_e^e$ 。最终的卡特兰数为 $\prod_e P_e$，其中内层和外层乘积均通过平衡乘积树计算。为确定能生成恰好有 1,234,567,890 个十进制位的卡特兰数的特定输入 n，我们反转了由斯特林公式 [3], 推导出的渐近近似。

$$C(n) \sim 4^n/(n^{3/2}\sqrt{\pi}).$$

通过求解超越方程,

$$\log_{10} C(n) \approx n \log_{10} 4 - \frac{3}{2} \log_{10} n - \frac{1}{2} \log_{10} \pi \approx 1,234,567,889$$

确定数值 $n = 2$，050，572，903 是一个有效候选值（通过计算 $\log C(n) + 1$ 的近似值进行验证）。这一预计算使我们能够精确锁定结果的数量级，从而确保得到一个有趣的结果。

这种方法实现了对 C(2，050，572，903) 的精确计算，其计算规模已证明超出 SageMath [7] 和 PARI/GP[6] 等通用软件的能力范围。通过提供因式分解文件和密码学哈希，确保了结果可复现。这一成果不仅为已知最大精确卡特兰数树立了新基准，也验证了一种适用于任何基于阶乘的数列的模块化架构。此外，分析还指出了整数重构中的关键瓶颈，为改进任意精度算术库提供了具体基准。

本文介绍了数学基础（第 2 节）、算法细节（第 3 节）和实现方法（第 4 节），并为每个阶段提供了伪代码。第 5 节报告了对 $n = 2,050,572,903$ 进行计算的实验结果和资源使用情况。第 6 节讨论了瓶颈、未来改进方向以及时间和空间复杂度。本文最后介绍了验证策略，包括所提供的最终结果的 SHA-256 哈希，以便进行独立验证。此次计算的所有代码、数据和输出文件均可在 GitHub [10] 上获取。

2 数学基础

令 $v_p(m)$ 表示整数 m 的素因数分解中素数 p的指数（ p 进赋值）。勒让德公式 [1]指出，对于正整数 m 和素数 p,

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

这是一个有限和，因为 $\lfloor m/p^k \rfloor = 0$ 对于 $p^k > m$。该卡特兰数的 p-进赋值可表示为:

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

因此，每个素数 $p \le 2n$ 贡献一个指数 $e_p = v_p(C(n))$，且 $C(n) = \prod_{p \le 2n} p^{e_p}$。注意，$v_p(n+1)$ 仅当素数整除 $n+1$ 时才非零。由于已知 $C(n)$ 是一个整数（例如，可由组合解释或库默尔定理 [2] 得出），因此所有按上述方式计算的 e_p 都是非负的。使用这些指数可以避免构造巨大的阶乘，如 $(2n)!$、$n!$ 等。

3 算法概述

3.1 阶段1：因式分解生成（catalan.py）

所有素数 $p \le 2n$均通过并行分段埃拉托斯特尼筛法[8]生成。该筛法使用 Python 实现，首先找出不超过 $\le \sqrt{2n}$的素数作为种子；随后并行处理互不重叠的区间，以最大限度地提高多核架构上的吞吐量。对于找到的每个素数 p，使用勒让德公式计算其 p-进赋值

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

。然后 $e_p = a_p - 2b_p - c_p$。对于素数 $n < p \le 2n$，项 $v_p((2n)!) - 2v_p(n!)$可简化为 1 或 0（因为当且仅当 p在二项式 $\binom{2n}{n}$中恰好出现一次时，$\lfloor 2n/p \rfloor - 2\lfloor n/p \rfloor$ 才为 1）；代码会自动处理这种情况。脚本使用素数列表通过试除法对$n+1$ 进行因式分解，从而得到 c_p。

随后，根据计算出的指数 e_p对素数进行分组。令 G_e 表示所有指数为 e 的素数集合。在所给出的实现中，生成的因式分解数据会按照以下结构序列化为文本文件（格式为 `catalan [n] factorization.txt`；[n]表示 n 的值）：

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

每个区块列出具有相同指数 E 的所有素数。与此同时，代码会累积 $\log_{10} C(n)$ 的高精度近似值，以估算位数（随后将其作为元数据输出）。概述这一阶段的伪代码见算法 1。

算法 1 Catalan(n) 的因式分解生成（在 `catalan.py` 中） 要求：整数 n。1：通过并行分段筛计算所有素数 $\le 2n$ 。2：使用素数 $\le \sqrt{n+1}$ 对 $m = n+1$ 进行试除法因式分解。3：对于每个素数 ~~$p \le$~~ $2n$ ，执行 4：计算 $a = \sum_{k\ge 1} \lfloor 2n/p^k \rfloor$。5：计算 $b = \sum_{k\ge 1} \lfloor n/p^k \rfloor$。6：设定 $e = a - 2b$。7：如果 p 能整除 $n+1$ ，且指数为 c ，则 8：设定 $e = e - c$。9：结束如果。10：如果 $e > 0$ ，则 11：将 p 添加到组 G_e。12：结束如果。13：结束循环。14：将每个组 G_e 写入 `catalan` n `factorization.txt`。15：估算并输出 $C(n)$ 的位数。

- -

3.2 阶段 2：重构（`catalanreconstructor.py`）

在重构阶段，读取预先计算的因式分解数据，以构造 $C(n)$。文件按指数分组逐组处理。对于每个指数 e 及对应的素数集合 G_e，基本乘积

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

会被计算出来。此操作采用内存高效的平衡乘积树算法，通过递归配对并相乘操作数，最大限度地缩短中间值的位长度。随后，使用经过优化的任意精度幂运算例程（`gmpy2` [9]）计算项 $P_e = Q_e^e$。最后，通过平衡乘积树再次汇总部分结果，得到卡特兰数 $\prod_e P_e$。最终的整数会被序列化到二进制文件中，并可选择生成十进制文本输出以供验证。使用二进制序列化，是为了避免在中间步骤中进行十进制转换和文本输入输出所带来的巨大开销。

算法 2 概述了这一过程，其中 BalancedProductTree(S) 表示对集合 S 中的元素进行递归分治乘法。鉴于集合 G_e 和中间列表 $\mathcal{P}$ 的规模，操作会分批处理为数据块（通常每次处理 10^5 个元素）。采用混合策略，通过定期进行内存回收并将数据卸载到磁盘，严格限制组合这些部分乘积时的内存用量。

算法 2 用于 Catalan(n) 的整数重构（在 `catalanreconstructor.py` 中）**要求：** 来自文件的因式分解组 G_e。1: 初始化列表 $\mathcal{P} = []$。2: **对于**素数列表 G_e 中的每个指数 e **执行**3: 计算 Q_e =BalancedProductTree(G_e)。4: 使用快速幂计算 $P_e = Q_e^e$。5: 将 P_e 追加到 $\mathcal{P}$。6: **结束循环**7: 计算 $C(n)$ = BalancedProductTree($\mathcal{P}$)。8: 将 $C(n)$ 写入 `catalan` n `reconstructed.bin`。

4 实现与可复现性

代码使用 Python 3 和标准库实现。主要文件如下：

- `catalan.py`：阶段 1 因式分解生成。
- `catalanreconstructor.py`：阶段 2 整数重构。

后者使用 `gmpy2` 库（该库封装了 GMP [5]）进行多精度算术运算。前者使用 `multiprocessing` 进行并行筛选区段处理。

要复现该结果，应运行 `catalan.py`，并传入 $n = 2,050,572,903$，这将生成 `catalan 2050572903 factorization.txt`。然后对该因式分解文件运行 `catalanreconstructor.py`，以生成 `catalan 2050572903 reconstructed.bin`。这些输出文件可供参考。所有脚本都包含参数（区段大小、分块大小），并设有适用于本次运行的默认值；针对其他硬件，可调整这些参数。关于我的计算机规格的更多信息，请参阅第 5 节。

5 结果

两个阶段在指定硬件上依次执行。表 1 总结了关键性能指标。值得注意的是，输入 $n = 2,050,572,903$ 得到了一个恰好有 1，234 个 567 个 890 十进制位数的卡特兰数（符合预期）。最终二进制结果文件的大小为 512,643,220 字节（$= 512.643$ MB 十进制制式，$= 488.895$ MiB）。总墙钟时间为 1549.14 秒，其中阶段 1 耗时 179.37 秒，阶段 2 耗时 1369.77 秒（使用 `time` 测量）。阶段 2 的峰值内存使用量约为 4 GB；相比之下，阶段 1 的内存消耗可以忽略不计。

这些结果表明，使用商用硬件计算十亿位数规模的卡特兰数是可行的，尤其考虑到在解释型语言中进行数值求值时不可避免的开销。为衡量所提出算法在内存和时间效率方面相较于开源求值工具的表现，表 2 测量了不同卡特兰数求值的运行时间。此数据使用 Sage 中的 `timeit` 函数生成，该函数会在多次试验中测量墙钟时间；我在本地机器上将报告的墙钟时间测量五次并取平均值。

表 1： $n = 2$，050，572，903 的计算统计数据。

统计量	值
n	$2,050,572,903$
$C(n)$ 的十进制位数	1，234，567，890
$C(n)$ 的位长度	4,101, 145, 759
结果文件大小	488.9 MiB
输出的 SHA-256	dac68f4ee35db8e9400e68bd6140e6cbccec6fb8ce81059318400e2c44e45ae4
阶段 1 运行时间（墙钟时间）	179.37 秒
阶段 2 运行时间（墙钟时间）	1369.77 秒
总墙钟时间	1549.14 秒
峰值内存（阶段 2）	4 GB
硬件	AMD Ryzen 9 4900HS，16 GB 内存，6 GB 图形处理器（未使用）
OS	Windows
软件	Python 3.11.9，gmpy2

表 2： $C(n)$ 的运行时间。

n的值	所提算法（s）	SageMath [7] 在本地运行（s）
$n = 10^6$	2.923 秒	0.378 秒
$n = 10^7$	6.817 秒	7.30 秒
$n = 10^8$	62.835 秒	134.0 秒

此外，还使用 PARI/GP（v2.15） [6]进行了比较分析。对于较小的 n 值，PARI/GP 的性能更佳，但在 $n \approx 1.34375 \cdot 10^9$ 处遇到了内存耗尽错误。通用 CAS 中的标准递归实现或基于阶乘的实现通常需要将中间值载入内存，而这些值超出了测试环境 16 GB 的容量，这凸显了对于如此大规模的输入，采用基于磁盘的素数指数法的必要性。

6 讨论

两阶段流程——先因式分解，再进行重构——是该算法在这一规模下可行的关键。阶段 1，即因式分解，可在内存要求不高的机器上执行，因为其主要任务是筛选素数并计算勒让德公式中的指数。无论从计算时间还是内存来看，主要瓶颈都是重构阶段。组合素数幂所需的大整数乘法占用了大部分运行时间，也决定了极大规模计算的硬件要求。

重构期间，磁盘输入输出也可能成为限制因素，尤其是在将中间结果转存到磁盘以管理内存时。因此，为缓解这一潜在瓶颈，强烈建议使用吞吐量高的存储设备，例如非易失性内存 express 或其他固态硬盘，而非传统机械硬盘。最终，阶段 2 的性能受可用内存容量以及底层多精度算术库效率的限制。

6.1 时间复杂度

- **筛法：** 枚举不超过 $2n$ 的素数，工作量约为 $\mathcal{O}(n \log \log n)$。筛法可从并行化中显著受益。

- **指数计算：** 对于每个素数 p，通过勒让德公式计算其指数需要进行一系列整除取底运算。对于本工作所针对的较大 n 值，这一步的开销远小于重构阶段。

- **重构：** 这一阶段的主要开销是大整数乘法。设 B 为最终整数的位长度，其中 $B = \Theta(n \log n)$。若 $M(B)$ 表示两个 B 位整数相乘的开销，则重构开销约为 $\mathcal{O}(M(B) \log P)$，其中 P 为不同素因子的数量。实际中，采用 GMP 的 [5] 高级乘法算法后，其渐近复杂度表现为 $\Theta(n(\log n)^2)$ 位运算。

6.2 空间复杂度

峰值内存使用量取决于阶段 2 的平衡乘积树归约过程中保存在内存中的最大中间整数。该整数的大小（因而空间复杂度）由最终卡特兰数的位长度决定，为 $\Theta(n \log n)$ 位。筛法生成的素数需要占用 $\mathcal{O}(n/\log n)$ 个机器字的存储空间。双阶段设计通过在开始重构前将完整因式分解结果存储到磁盘上，以增加磁盘输入输出为代价，有效降低了峰值内存需求。

6.3 可推广性

虽然本文聚焦于卡特兰数列，但所提出的架构具有模块化特性。具体来说，重构工具 (`catalanreconstructor.py`) 是一个通用引擎，可用于计算素数幂的乘积，而不受指数组合来源的限制。因此，该方法可扩展到任何由阶乘乘积之比定义的整数序列，包括二项式系数 $\binom{n}{k}$、多项式系数和超阶乘。

形式化地，设 $\mathcal{F}$ 表示可表示为以下形式的组合量类别：

$$\mathcal{F} = \frac{\prod_{i=1}^{k}(a_i!)}{\prod_{j=1}^{m}(b_j!)}$$

其中 $a_i, b_j \in \mathbb{Z}^+$。任何 $N \in \mathcal{F}$ 的素因数分解都可以通过扩展勒让德公式来确定。根据 p-进赋值的线性性质 v_p，因式分解中任意素数 p 的指数为：

$$v_p(N) = \sum_{i=1}^{k} v_p(a_i!) - \sum_{j=1}^{m} v_p(b_j!)$$

代入勒让德公式 $v_p(n!) = \sum_{r=1}^{\infty} \lfloor \frac{n}{p^r} \rfloor$，我们得到：

$$v_p(N) = \sum_{i=1}^{k} \sum_{r=1}^{\infty} \left\lfloor \frac{a_i}{p^r} \right\rfloor - \sum_{j=1}^{m} \sum_{r=1}^{\infty} \left\lfloor \frac{b_j}{p^r} \right\rfloor$$

随后，将该值 N 重构为所有相关素数的乘积 s:

$$N = \prod_{p \le \max(\max_i a_i, \max_j b_j)} p^{v_p(N)}$$

算法的阶段 1 可通过简单修改指数求和逻辑，推广至 $\mathcal{F}$ 中的任意成员。请注意，根据整性，计算得到的指数 $v_p(N)$ 均为非负数；否则，负指数意味着该表达式不是整数。

阶段 2 保持不变。假设使用基于快速傅里叶变换的乘法（例如 GMP 在处理大操作数时使用的算法），重构的时间复杂度取决于最终整数的位长度，详见第 6.1 节。

7 结论

本文提出了一种可复现且内存高效的框架，用于计算超大规模组合数，并通过精确计算 $C(2,050,572,903)$ 展示了该框架。该算法完全在素数指数域中运算，从而避开了通用计算机代数系统中标准递归实现所受的内存限制。

如第 6.3 节所述，这种架构并不限于卡特兰数列；解耦的重构引擎可推广至任何由阶乘比定义的整数。这为使用商用硬件在更广泛的计数组合学领域开展高精度研究提供了一条可行途径。所提供的源代码和因式分解数据可用于独立验证，也可作为优化大整数算术库的基准。

最后，卡特兰数有许多组合学上的解释；其中一种经典解释得出了一个具体结果：

> **推论**。 对一个有 *2,050,572,905* 条边的凸多边形进行三角剖分，其不同方法的数量恰好等于本文标题中给出的整数 *[4]*。

参考资料

[1] A.-M. Legendre，数论 ´论，Duprat，巴黎，179 8..[2] E. E. Kummer，《关于一般互反律的补充定理》，纯粹与应用数学杂志 ¨，第 44 卷，第 93–146 页，1852 年。[3] J. Stirling，《微分法：无穷级数求和与插值论》，伦敦，1730 年。[4] R. P. Stanley，《卡特兰数》，剑桥大学出版社，2015 年。[5] T. Granlund 等，《GNU MP：GNU 多精度算术库》，6.3.0 版，2023 年。可查阅：`https://gmplib.org/`[6] PARI 团队，《PARI/GP 2.15 版》，波尔多大学，2024 年。可查阅：`http://pari.math.u-bordeaux.fr/`

[7] Sage 开发者，《SageMath，Sage 数学软件系统（10.0 版）》，2023 年。可查阅：https://www.sagemath.org[8] R. Crandall 和 C. Pomerance，《素数：计算视角》，第 2 版，Springer，纽约，2005 年。[9] C. Hart 等，《gmpy2：连接 GMP、MPFR 和 MPC 的 C/Python 接口》，计算中使用的版本，2025 年。可查阅：https://gmpy2.readthedocs.io/[10]M. Ramani，“math-portfolio”，GitHub 仓库，提交 c5af71cfc56299308e9d65d347be01fd5a0a0a8c，2025 年。可查阅：https://github.com/Mahesh-Ramani/math-portfolio/tree/c5af71cfc56299308e9d65d347be01fd5a0a0a8c/catalans