nth: Open-Source Generic Combinatorial (Un)ranking via the Symbolic Method

Marcos Tomaszewski, Gustavo Zambonin[0000-0002-6962-9347], Jean Everson Martina[0000-0003-4104-1741], and Ricardo Custódio[0000-0001-9611-5694]

Universidade Federal de Santa Catarina, Florianópolis, SC, 88040-900, Brazil
{marcos.tomaszewski,gustavo.zambonin}@posgrad.ufsc.br,
{jean.martina,ricardo.custodio}@ufsc.br

Abstract. We present `nth`, an open-source C library for generic ranking and unranking of combinatorial structures. It accepts formal specifications in a language compatible with Maple's `combstruct`, covering Union, Product, Set, Sequence, Cycle, PowerSet, substitution, pointing, and the labeled boxed product. Counting and exact-size uniform random generation (drawing) work in both labeled and unlabeled settings; ranking and unranking follow the framework of Martínez and Molinero (Random Structures & Algorithms, '01), covering all operators except unlabeled Cycle. Beyond generation, `nth` recovers an explicit generating function (rational, algebraic, holonomic, closed form, or an implicit functional system) for each specification it solves, verified against its own counting sequence. Python and SageMath bindings expose the full API, and correctness is validated against an OEIS-keyed corpus of thousands of specifications, correcting and extending the legacy Encyclopedia of Combinatorial Structures.

Keywords: Combinatorial enumeration · Analytic combinatorics · Open-source software

1 Introduction

A short history of automatic combinatorics software. The automatic treatment of decomposable structures began in the Maple family. The $\Lambda\upsilon\Omega$ (LUO) system mechanized average-case analysis over combinatorial specifications [7]. Its series machinery became `gfun` [20], and its random-generation calculus, namely the recursive method of Flajolet, Zimmermann, and Van Cutsem with the boustrophedonic order [9], became Gaïa [24]. Both were research code of the INRIA Algorithms Project, first distributed through the Maple share library; Zimmermann and Murray consolidated them into `combstruct` [6], later incorporated into the standard Maple library. Mishna later extended it with attribute grammars for automatic complexity analysis [16] and experimented with ranking and unranking; that work did not cover all constructions and was never merged. `combstruct` ships with every Maple release to this day, but it is proprietary, and generic ranking and unranking were never added.

A second line of development targeted MuPAD, a then-independent computer algebra system: its `CS` package [3] (1998) grew into MuPAD-Combinat [10], the closest predecessor to `nth`, providing generic counting, unranking, and random generation from a grammar. Its fate motivates our work: when MathWorks acquired MuPAD in 2008, the community migrated part of the functionality to Sage-Combinat, but the grammar-driven generation was not carried over.

A third, standalone branch targets approximate-size Boltzmann sampling [4] rather than exact-size generation, from GenRGenS [19] through Paganini [2] to CombOL [5], a Rust library with Python bindings that enumerates and samples decomposable structures from a grammar, our contemporary point of comparison in Section 4. Hence, every prior vehicle for generic combinatorial (un)ranking was either closed-source, or open but dependent on a closed platform that eventually discarded it.

Contributions. We fix this gap by providing a modern, permissively licensed open-source tool called `nth`, which implements combinatorial (un)ranking algorithms that work generically from a symbolic-method grammar, covering several labeled and unlabeled admissible operators (except for unlabeled Cycles, a long-standing open work). Using `nth`, we

(i) audit and extend the legacy Encyclopedia of Combinatorial Structures into a collection of over 3000 validated specifications indexed by the On-Line Encyclopedia of Integer Sequences (OEIS) A-number, correcting spurious references and adding grammar–sequence pairs discovered by fuzzing;
(ii) recover a verified generating function (g.f.) for every specification in this corpus, a capability related to the automatic-series line of `gfun` [20] and attribute grammars [16]; and
(iii) measure the performance of counting and exact-size random generation against Maple's `combstruct` and CombOL [5], achieving faster drawing of random combinatorial objects for several parameter sets.

2 Preliminaries

We mainly use definitions from the Analytic Combinatorics book [8]. We write $[m]$ for $\{0, 1, \ldots, m\}$, ∂_z for the derivative in z, and ϕ for Euler's totient function.

Definition 1 ([8, Definition I.1]). *A* combinatorial class *is a finite or denumerable set $\mathcal{C}$ equipped with a size function $|\cdot| : \mathcal{C} \to \mathbb{Z}_{\geq 0}$ such that the number c_n of elements of each size n is finite.*

The symbolic method [8] translates a combinatorial specification, i.e., a *grammar* of equations over atomic objects and admissible constructions, into functional equations on the generating functions (g.f.s) of the classes it defines. An *unlabeled* class uses an ordinary g.f. (OGF) $C(z) = \sum_{n\geq 0} c_n z^n$ with $c_n = |\mathcal{C}_n|$; a *labeled* class, whose size-n objects carry a bijection of labels $\{1, \ldots, n\}$ onto their atoms, uses an exponential g.f. (EGF) $C(z) = \sum_{n\geq 0} c_n z^n/n!$.

Definition 2 ([12, Section 2.1]). *Let $\mathcal{C}_n$ be the set of size-n objects of a combinatorial class, with $c_n = |\mathcal{C}_n|$ finite. A* ranking function *is a bijection $\mathbf{R} : \mathcal{C}_n \to [c_n - 1]$; it defines a total order $\prec$ on $\mathcal{C}_n$ by $s \prec t$ if and only if $\mathbf{R}(s) < \mathbf{R}(t)$. The* unranking function *is the inverse bijection $\mathbf{U} = \mathbf{R}^{-1}$.*

Martínez and Molinero (MM) [14] showed that both functions can be derived generically from any labeled specification by recursively partitioning rank spaces at each grammar node, distributing label sets via binomial coefficients and computing subset ranks with the combinatorial number system [11]. Molinero and Vives later extended the derivation to the unlabeled setting [18], where there are no labels to distribute and components are canonicalized instead. Theorems 1 and 2 state the generating-function translations of the admissible constructions.

Theorem 1 ([8, Theorem II.1, II.3 and II.5]). *Let A, B, C be labeled classes with EGFs $\hat{A}, \hat{B}, \hat{C}$ and $C(0) = 0$. Then (i)* $\mathsf{Union}(A, B)$ *has EGF $\hat{A} + \hat{B}$; (ii)* $\mathsf{Prod}(A, B)$ *has $\hat{A}\hat{B}$; (iii)* $\mathsf{Sequence}(A)$ *has $\frac{1}{1-\hat{A}}$; (iv)* $\mathsf{Set}(A)$ *has $e^{\hat{A}}$; (v)* $\mathsf{Cycle}(A)$ *has $\log \frac{1}{1-\hat{A}}$; (vi)* $\mathsf{Subst}(B, C)$ *has $\hat{B}(\hat{C}(z))$; (vii)* $\mathsf{Point}(A)$ *has $z\,\partial_z \hat{A}$; and (viii) the boxed product* $\mathsf{BoxProd}(B, C)$*, the labeled product restricted to objects whose smallest label lies in B (requiring $B_0 = 0$), has $\int_0^z (\partial_t \hat{B})\,\hat{C}\,dt$, with coefficients $A_n = \sum_{k=1}^{n} \binom{n-1}{k-1} B_k\, C_{n-k}$.*

Theorem 2 ([8, Theorem I.1 and I.4]). *Let A, B, C be unlabeled classes with OGFs A, B, C, $a_n = [z^n]A(z)$, and $C(0) = 0$. Then (i)* Union, Prod, *and* $\mathsf{Sequence}$ *translate as in the labeled case; (ii) the multiset* $\mathsf{Set}(A)$ *has OGF $\prod_{n\geq 1}(1-z^n)^{-a_n}$; (iii) the set of distinct elements* $\mathsf{PowerSet}(A)$ *has $\prod_{n\geq 1}(1+z^n)^{a_n}$; (iv)* $\mathsf{Cycle}(A)$ *has $\sum_{k\geq 1} \frac{\phi(k)}{k} \log \frac{1}{1-A(z^k)}$; and (v)* Subst *and* Point *translate as in the labeled case; the boxed product has no unlabeled counterpart.*

3 The Tool

Specification language. `nth` accepts specifications, lists of possibly mutually recursive named equations, as text strings, in a syntax that extends the grammar of Maple's `combstruct` package while remaining interoperable with it. The atom `z` has size 1 and `Epsilon` size 0; the constructions translate as in Theorems 1 and 2, with cardinality restrictions (`=`, `>=`, `<=`, `>`, `<`, and the interval `a..b`) on Set, Sequence, PowerSet, and Cycle.

We also implement extensions, such as arbitrary interval cardinalities, aliases (MSet, PSet, Box), implicit lowercase atoms, indexed identifiers, comments, and the user-level Point and BoxProd. We keep the `combstruct` operator names for this compatibility: Set is the multiset MSet of Flajolet and Sedgewick [8] and PowerSet their PSet. The two differ only in the unlabeled setting, and labeled PowerSet collapses to Set [8, Page 103], which is honored by `nth`.

Architecture. `nth` is written in C and depends only on FLINT [21], using its `fmpz`, `fmpz_poly`, and `fmpq_poly` types throughout so that coefficients and ranks stay exact regardless of magnitude. We design a minimal C API which takes as input strings representing specifications, and outputs exact integers or object strings, so any language with a C foreign function interface (FFI) can interact with it. Ranking and unranking are derived from the grammar alone, with no hand-crafted code per structure; a solved specification is a persistent context whose coefficient arrays and (optionally memoized) counting tables are computed once and shared across queries.

We organize the tool into four modules: (i) a hand-written recursive-descent parser, with a well-foundedness check that rejects ill-founded specifications; (ii) a counting module for generating-function coefficients; (iii) a ranking/unranking module implementing the MM bijections; and (iv) a generating-function recovery module, all exposed through a command-line interface helper and through the C API. The operations mirror the framework: computing counting sequences, ranking and unranking, uniform random generation (drawing, via unranking a uniform random rank), exhaustive enumeration, round-trip self-tests and generating-function recovery.

Every operation works in the labeled and unlabeled settings, under either unranking order (lexicographic or boustrophedonic [9]) where applicable. We also provide Cython bindings that expose the whole API to Python, with FLINT integers converting to native Python integers. Analogously, we also give a thin SageMath wrapper which returns Sage `Integer`s.

Counting. `nth` computes the coefficient array $[c_0, \ldots, c_N]$ for every named symbol up to a bound N, iterating coefficient evaluation to a fixed point and applying the translations of Theorems 1 and 2. We make use of FLINT's truncated power-series arithmetic: unlabeled coefficients as big-integer polynomials (`fmpz_poly_{mullow,inv_series,compose_series}`), labeled ones by rational arithmetic on the EGF coefficients $c_n/n!$, whose `fmpq_poly_mullow` product realizes the binomial convolution for Prod. Labeled Set and Cycle map to `fmpq_poly_{exp_series,log_series}`, avoiding term-by-term sums.

Ranking and unranking. We implement the MM framework [14] for labeled structures and the Molinero–Vives extensions [18] for all unlabeled operators except Cycle. Both are recursive: at each node the rank space is partitioned according to the construction and the algorithm recurses into the appropriate sub-problem. Writing x_n for the number of size-n objects of a class X, $\mathsf{Union}(A, B)$ splits $[a_n + b_n - 1]$ into $[a_n - 1]$ and its complement; $\mathsf{Prod}(A, B)$ partitions by left-component size k with offset $\sum_{j<k} a_j b_{n-j}$ and, when labeled, a binomial label draw; and $\mathsf{Sequence}(A)$ partitions by the number of components and then by their sizes.

In the labeled setting the unordered constructions reduce to one device, the boxed product of Theorem 1: distinguishing the component that carries the globally minimum label breaks their symmetry and yields an ordered decomposition

partitioned like Prod. A nonempty $\mathsf{Set}(A)$ decomposes into the min-carrying component followed by the remaining set; $\mathsf{Cycle}(A)$ into the min-carrying component followed by the rest of the cycle in order, a $\mathsf{Sequence}(A)$ tail; and the user-level $\mathsf{BoxProd}(B, C)$ exposes the device itself, with the minimum label pinned to B.

All three partition by the size k of the distinguished component: with a_k counting the component and t_{n-k} the tail, the block has weight $\binom{n-1}{k-1} a_k\, t_{n-k}$, and the rank within it decomposes as $r = r_{\mathrm{sub}}\, a_k\, t_{n-k} + r_A\, t_{n-k} + r_T$, where $r_{\mathrm{sub}} \in [\binom{n-1}{k-1} - 1]$ is the combinatorial-number-system rank of the chosen label subset [11]. Then, $r_A \in [a_k - 1]$ ranks the component, and $r_T \in [t_{n-k} - 1]$ the tail. Labeled PowerSet coincides with Set, as noted above.

In the unlabeled setting the unordered constructions canonicalize instead: a shared multiset decoder enumerates every component object of size at most n into one global index table and decodes a $\mathsf{Set}(A)$ as a monotone sequence of component indices. $\mathsf{PowerSet}(A)$ uses a similar decoder with repetition forbidden. Since this table holds one entry per component object, we store it up to a fixed cap of 2^{24} entries to prevent out-of-memory errors; for classes whose component counts grow past it, `nth` yields a documented error code.

For $\mathsf{Subst}(B, C)$, we unrank by partitioning the size-n rank space by the number k of atoms of the B-shape β: each block has size $B_k \cdot W_k$, where W_k counts the k C-objects of total size n [8, Definition I.14]. We recover β and the collection, then graft the C-objects into β's atom slots behind explicit boundary markers, so the encoding stays injective even for recursive B; ranking peels them back out and recombines the ranks. Finally, $\mathsf{Point}(A)$ has rank space $n\, a_n$ split as $r = r_A \cdot n + j$, distinguishing one atom of the object at r_A. For the iterated pointing Θ^k, we record the order of the marks, so it can be appropriately ranked and unranked.

Generating-function recovery. `nth` recovers an explicit generating function from its own exact counting series, with no computer-algebra system in the loop. The methodology is standard guessing [20]: the cascade tries, in order,

(i) rational reconstruction via Berlekamp–Massey [15];
(ii) an exact closed form by symbolic-method transfer rules, including Pólya sums for restricted unlabeled constructions [8];
(iii) an algebraic minimal polynomial via an integer Hermite–Padé approximation [1];
(iv) an implicit functional-equation system for recursive rules; and
(v) a holonomic ODE guess [20].

We accept a candidate only if it reproduces the counting sequence with the number of unknowns bounded by half the available terms, so that the surplus terms act as independent checks against overfitting a short prefix. Of course, we claim only that the result is verified, not formally proven.

Correctness. We initially tested `nth` against the legacy Encyclopedia of Combinatorial Structures data [13], which ships each specification with an OEIS reference, by recomputing every referenced sequence against the full OEIS term

list. Out of its 1075 entries, two record no usable reference and five cannot be compared; of the rest, 1018 agree up to OEIS's recorded offset and 50 do not. Most inconsistencies are genuinely incorrect references, where the specification counts a different sequence altogether (e.g., ECS#112 against A003416).

The rest split into sign conventions, where a grammar can only produce the absolute values of a signed OEIS sequence (e.g., ECS#75 against A007113), and grading differences, where the specification measures size by a different parameter than the OEIS indexing, so the sequences align only after dropping interleaved zeros or reindexing by a stride (e.g., ECS#55, whose counts are A000108 interleaved with zeros). The generating functions stored in the encyclopedia are also re-expanded with PARI/GP and checked against `nth`'s counts.

Beyond the audit, we also fuzz the library through one further operation it exposes: a generator of random well-founded specifications. We retain the generated grammars whose counts match an OEIS sequence, confirming each match against the full stored sequence and attaching the g.f. `nth` derives. This process grows the corpus to over 3000 entries, each validated by its independently curated OEIS terms.

Entries in the corpus are indexed by the OEIS number of the g.f. coefficient sequence. Each entry is checked on four levels: its counting sequence must equal the referenced OEIS terms; ranking and unranking must round-trip, over ranks sampled across the whole rank space, in both the lexicographic and boustrophedonic orders; distinct ranks must decode to distinct objects; and the stored generating function must be reproduced by the generating-function module.

We also conformance-test the grammars for the specification language and the object format against the C parser and serializer, and each entry carries its `combstruct` form, which must parse and count identically; the corpus thus doubles as a parser-compatibility oracle. Entries using a feature `nth` does not implement are skipped for the ranking levels but still checked for counting. The audit, fuzzing, and benchmark drivers of this section and Section 4 are distributed as a companion repository archived with the tool [23].

4 Experiments

Next, we compare the performance of counting and exact-size uniform random generation across `nth`, Maple's `combstruct`, and CombOL. We justify our experimental setup and choice of combinatorial classes in more detail below.

Test machine & source code. To reproduce the experiments discussed below, we provide source code and accompanying documentation [22,23]. All benchmarks are run in an AMD Ryzen™ 7 9700X @ 3.8GHz in a GNU/Linux environment (kernel 7.0.13), `gcc` 16.1.1 and FLINT 3.6.0. Executables are compiled with the `-O3 -march=native -mtune=native` set of optimization flags. We disable any simultaneous multithreading and dynamic frequency scaling technologies, and set the governor to `performance`. We use the `cpuset` kernel feature to isolate a CPU core to exclusively run the desired benchmarks, without intervention from

the scheduler, and disable address space layout randomization (ASLR) to reduce attempts from the operating system to cache certain memory pages.

Combinatorial class selection. We take all examples in the `combstruct` documentation (ten labeled, ten unlabeled), plus two algebraic tree classes supported by CombOL (plane binary and Motzkin trees), so that every backend is exercised. The labeled classes are nonplane, plane binary, plane general, and nonplane ternary trees, permutations, set partitions, hierarchies, three-balanced hierarchies, surjections, and functional graphs; the unlabeled ones are integer partitions (with and without repetition), binary sequences, necklaces, rooted, nonplane binary, and nonplane ternary trees, hierarchies, mapping patterns, and 2–3 trees.

This selection generally mirrors the classes used by Molinero in the experimental evaluation of the original framework [17, Table 5.1], which covers binary and Motzkin trees, functional graphs, integer partitions, surjections, and integer compositions, in both settings where admissible. Each class is measured at sizes $n \in \{10, 20, 50, 100, 200, 500\}$. In Figure 1, we report the geometric mean of medians of 5 cold runs for the counting operation, and of 33 warm runs for the draw operation, whose cost varies with the shape of the drawn object.

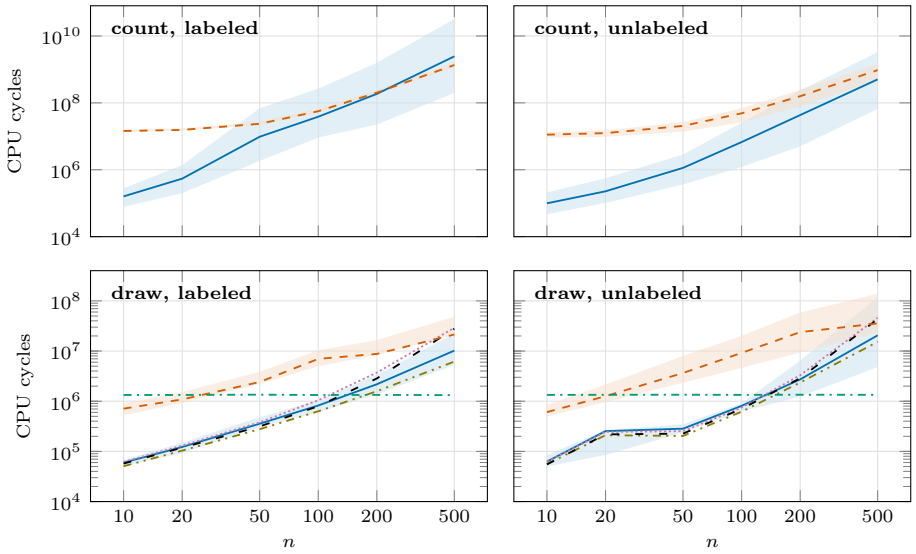

Fig. 1. Counting (top) and drawing (bottom) operations measured in CPU cycles (log scale) for labeled (left) and unlabeled (right) classes at various sizes n. Bold lines are geometric means over the classes each tool supports, and the shaded bands span the corresponding interquartile ranges. Colors and patterns: `nth` (blue, solid), `combstruct` (orange, dashed), CombOL (green, dash-dotted), `nth` without memoization (pink, dotted), `nth` under boustrophedonic order with memoization (olive, dash-dot-dotted) and without (black, loosely dashed). For the last three, the spread is similar to `nth` and is thus not shown.

Counting performance. Counting from scratch, i.e., with parse and solve included on the `nth` side and a fresh kernel per trial on the Maple side (kernel startup excluded), exhibits two regimes (Figure 1, top row). At small sizes the cost is dominated by setup, and `nth` is two orders of magnitude faster than `combstruct` (geometric mean 102× at $n = 10$). As n grows, both tools converge to the cost of arbitrary-precision polynomial arithmetic, and the gap narrows to 1.1× at $n = 500$. The spread around the mean is wide: non-recursive specifications such as binary sequences or permutations remain two to three orders of magnitude faster in `nth` at $n = 500$, while deeply recursive ones (labeled nonplane trees or functional graphs) invert the ratio, down to 0.01×, since the fixed-point solve repeats full-series operations at every iteration.

Random generation. We measure the performance of drawing in `combstruct`, `nth` (both exact-size) and CombOL (approximate-size via Boltzmann sampling). In the case of `nth`, this is done via unranking. We also consider the performance of `nth` with and without pre-computing counting tables, and for lexicographic and boustrophedonic orders. Figure 1 (bottom row) summarizes the results. On warm tables, `nth` holds a geometric-mean advantage of 8–12× over `combstruct:-draw` up to $n = 200$, settling at 3.0× at $n = 500$.

Disabling memoization makes `nth` draw a geometric-mean 2.6× slower at $n = 500$, and up to 26× slower on individual classes; the advantage over `combstruct` above thus rests in part on reusing work across draws from the same solved context. CombOL is size-independent at roughly 1.3 million cycles per object, so exact-size unranking is cheaper up to n between 100 and 200, and approximate-size sampling wins beyond, the expected trade-off between the two models. Some unlabeled classes (hierarchies, nonplane binary, and rooted trees) exceed the (multi)set rank-table cap past small sizes.

The choice of order matters in `nth`. Considering the boustrophedonic order, at $n = 500$ it is a geometric-mean 1.5× faster than the lexicographic order, with the gain concentrated on recursive tree classes (1.7–3.6× on hierarchies, nonplane, plane binary, and plane general trees, functional graphs, and Motzkin trees). These are the classes whose split positions spread over the whole size range, which is exactly where interleaving the probes from both ends pays off; the advantage surfaces because the block weights are memoized, so a probe costs a lookup rather than a big-integer product. Indeed, with memoization disabled, the two orders are indistinguishable (geometric-mean 1.05× at $n = 500$). Classes whose cost lies outside the split scans are unaffected, and no class is more than 10% slower.

5 Conclusion

We have presented `nth`, an open-source C implementation of the Martínez–Molinero generic (un)ranking framework, covering several admissible labeled and unlabeled operators (except unlabeled Cycle), with counting, uniform generation, generating-function recovery, and Python/SageMath bindings. The package is

extensively tested and maintainable, and we release it under the MIT license. We expect that the library is useful for research in combinatorial ranking algorithms and combinatorics in general, serving as an alternative to (but interoperable with) proprietary systems.

Future work. Evidently, the main open problem is unranking unlabeled cycles, which calls for replacing the combinatorial number system with an encoding based on the integer partition lattice. On the integration side, the existing SageMath wrapper can be extended into a first-class interface to the `LazyPowerSeries` and `GenericCombinatorialSpecies` SageMath classes. The C API can be readily ported to Julia, Haskell, or even Maple itself. Formal verification of the generic algorithms (e.g., in Rocq or Lean) is another exciting avenue of research, strengthening the empirical round-trip guarantees.

Acknowledgments. We are grateful to the anonymous reviewers for their helpful comments and suggestions, particularly for pointing out that Marni Mishna worked on (un)ranking but that work was never added to `combstruct`. We thank Conrado Martínez, Lucia Moura and Daniel Panario for feedback of an early version of this work. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. G. Zambonin was partially supported by the Fundacao de Amparo a Pesquisa e Inovacao do Estado de Santa Catarina (FAPESC), Edital 62/2024, and by the National Operator of the Civil Registry of Natural Persons (ON-RCPN) of Brazil.

Disclosure of Interests. The authors have no competing interests to declare.

References

1. Beckermann, B., Labahn, G.: A Uniform Approach for the Fast Computation of Matrix-Type Padé Approximants. SIAM Journal on Matrix Analysis and Applications **15**(3), 804–823 (Jul 1994). `https://doi.org/10.1137/S0895479892230031`
2. Bendkowski, M., Bodini, O., Dovgal, S.: Polynomial tuning of multiparametric combinatorial samplers. In: Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO). pp. 92–106 (Jan 2018). `https://doi.org/10.1137/1.9781611975062.9`
3. Denise, A., Dutour, I., Zimmermann, P.: CS: a MuPAD package for counting and randomly generating combinatorial structures. In: Proceedings of the 10th Conference on Formal Power Series and Algebraic Combinatorics (FPSAC'98). pp. 195–204 (Jun 1998)
4. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Boltzmann Samplers for the Random Generation of Combinatorial Structures. Combinatorics, Probability and Computing **13**(4-5), 577–625 (Jul 2004). `https://doi.org/10.1017/S0963548304006315`
5. Eriksen, C.A.: CombOL: The Combinatorial Objects Library (2026), `https://gitlab.com/casbjorn/combol`
6. Flajolet, P., Salvy, B.: Computer algebra libraries for combinatorial structures. Journal of Symbolic Computation **20**(5), 653–671 (Nov 1995). `https://doi.org/10.1006/jsco.1995.1070`

7. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic average-case analysis of algorithms. Theoretical Computer Science **79**(1), 37–109 (Feb 1991). https://doi.org/10.1016/0304-3975(91)90145-R
8. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, 1st edn. (2009). https://doi.org/10.1017/CBO9780511801655
9. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science **132**(1), 1–35 (Sep 1994). https://doi.org/10.1016/0304-3975(94)90226-7
10. Hivert, F., Thiéry, N.M.: MuPAD-Combinat, an open-source package for research in algebraic combinatorics. Séminaire Lotharingien de Combinatoire **51** (2004), article B51z
11. Knuth, D.E.: The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. Addison-Wesley Professional, 1st edn. (2011)
12. Kreher, D.L., Stinson, D.R.: Combinatorial Algorithms: Generation, Enumeration, and Search. CRC Press, 1st edn. (1998)
13. Lumbroso, J.: Encyclopedia of Combinatorial Structures Dataset (2018), https://github.com/jlumbroso/encyclopedia-of-combinatorial-structures-data
14. Martínez, C., Molinero, X.: A Generic Approach for the Unranking of Labeled Combinatorial Classes. Random Structures & Algorithms **19**(3-4), 472–497 (Oct 2001). https://doi.org/10.1002/rsa.10025
15. Massey, J.L.: Shift-register synthesis and BCH decoding. IEEE Transactions on Information Theory **15**(1), 122–127 (Jan 1969). https://doi.org/10.1109/TIT.1969.1054260
16. Mishna, M.: Attribute grammars and automatic complexity analysis. Advances in Applied Mathematics **30**(1-2), 189–207 (Feb 2003). https://doi.org/10.1016/S0196-8858(02)00542-0
17. Molinero, X.: Ordered generation of classes of combinatorial structures. PhD thesis, Universitat Politècnica de Catalunya (Jul 2005)
18. Molinero, X., Vives, J.: Unranking Algorithms for Combinatorial Structures. International Journal of Applied Mathematics and Informatics **9**, 110–115 (2015)
19. Ponty, Y., Termier, M., Denise, A.: GenRGenS: software for generating random genomic sequences and structures. Bioinformatics **22**(12), 1534–1535 (Mar 2006). https://doi.org/10.1093/bioinformatics/btl113
20. Salvy, B., Zimmermann, P.: GFUN: a Maple package for the manipulation of generating and holonomic functions in one variable. ACM Transactions on Mathematical Software **20**(2), 163–177 (Jun 1994). https://doi.org/10.1145/178365.178368
21. The FLINT team: FLINT: Fast Library for Number Theory. Version 3.6.0 (2025), https://flintlib.org
22. Zambonin, G.: `nth`. Version 0.2.0 (Jul 2026), https://doi.org/10.5281/zenodo.21155805
23. Zambonin, G.: `nth-tools`. Version 0.2.0 (Jul 2026), https://doi.org/10.5281/zenodo.21155816
24. Zimmermann, P.: Gaïa: a package for the random generation of combinatorial structures. MapleTech **1**(1), 38–46 (1994)