

BIGGER WORLDS, NARROWER SEARCHES

Rethinking Scaling through Exact Aggregation and Sampling Thinning

Michael S Yang
Independent Researcher
yangofzeal@gmail.com

Keywords: computational scaling; exact aggregation; sufficient states; sampling thinning; integer arithmetic; reproducible computation.

Abstract. Problem size alone does not determine the work performed by a structured computation. This methodological note separates three quantities: the number of possible histories, the number of distinguishable computational states, and the number of samples in a declared feasible region. We give complete proofs for two independent public models. First, exact aggregation by equal partial sums preserves every binary selection count and a verifiable witness while replacing histories by sufficient states. For repeated unit weights, there are 2^n histories, $n + 1$ final states, and exactly $n(n + 1)$ candidate contributions in the implemented recurrence. Second, an integer grid in a rational interval has an exact, nonincreasing sample count as its scale parameter grows, with a sharp budget threshold. We also prove limits to both interpretations: state aggregation can remain exponential, and thinning a feasible region can change its optimum. A dependency-free Python artifact and independent-oracle tests make all examples reproducible. The contribution is a precise separation of scaling claims and their proof obligations, not a new general-purpose solver or a universal inverse-runtime law.

1 INTRODUCTION

The phrase “a larger problem” can refer to an input’s encoding length, its number of candidate histories, its reachable-state width, or an external scale parameter. These are different quantities. A claim that computation becomes smaller is meaningful only after the varying parameter, preserved task, and work metric have been specified.

Dynamic programming provides a classical example: histories sharing sufficient state need not be processed separately. Subset-sum dynamic programming and the distinction between polynomial and pseudopolynomial bounds are standard [1]. Generating functions provide the associated exact counting language [2]. An earlier manuscript by the present author also examined structured state widths in subset-sum computation [3]. The present note is self-contained; it neither requires that manuscript’s pruning rules nor assumes that a width cap preserves solutions.

Our central question is narrower and testable: *what exactly can become smaller, and what information survives?* We study lossless coalescence of equal states and exact thinning of a prescribed rational grid. The first preserves the requested output for a fixed input. The second compares different feasible grids and therefore requires an explicit qualification before any statement about solution preservation. Their elementary ingredients are established mathematics. The value of their joint treatment is an auditable account of work, correctness, counterexamples, and executable evidence.

2 EXACT EXPLORATION BY SUFFICIENT STATES

Let $w_1, \dots, w_n$ be nonnegative integers. Equal values and zero values are allowed; positions remain distinct. At layer i , a history is a vector $x \in \{0, 1\}^i$, with state

$$\sigma_i(x) = \sum_{j=1}^i w_j x_j. \quad (1)$$

Write $R_i = \{\sigma_i(x) : x \in \{0, 1\}^i\}$ and $W_i = |R_i|$. The multiplicity of state s is

$$c_i(s) = |\{x \in \{0, 1\}^i : \sigma_i(x) = s\}|. \quad (2)$$

Unreachable states have multiplicity zero. Initially, $R_0 = \{0\}$ and $c_0(0) = 1$.

Definition 1 (Exact aggregation). At a fixed layer, histories are equivalent precisely when their partial sums are equal. An aggregated state records its sum, total multiplicity, and one indexed witness. It does not record all histories individually.

Theorem 2 (Exploration and exact coalescence). *For every $i \geq 1$ and integer s ,*

$$c_i(s) = c_{i-1}(s) + c_{i-1}(s - w_i). \quad (3)$$

Consequently, aggregating equal partial sums after every expansion preserves all reachable sums, their exact multiplicities, and the existence of a witness for every reachable sum. Moreover,

$$\sum_{s \in R_i} c_i(s) = 2^i. \quad (4)$$

Proof. Partition histories of length i ending at s by their last bit. Those ending in zero are in bijection with histories of length $i - 1$ ending at s . Those ending in one are in bijection with histories ending at $s - w_i$. The two classes are disjoint even when $w_i = 0$, so their counts add. This proves (3).

The initialization is exact. Inductively, each old reachable sum contributes an exclusion candidate at s and an inclusion candidate at $s + w_i$. Every history appears in one of these two classes, and each candidate is realized by extending an old witness. Coalescing identical candidates adds their multiplicities and retains one of their valid witnesses. No attainable sum is discarded and none is introduced without a witness. Summing the recurrence doubles the previous total mass, giving (4) by induction. $\square$

The equivalence is valid because all histories in a layer face the same suffix of weights. Equal partial sums therefore have identical future attainable increments. If a later rule depended on the order of previous choices, their identities, or a constraint not encoded by the sum, this state description would require refinement. Coalescence is exact only relative to the stated output: sums, counts, and one witness, not enumeration of all histories.

Corollary 3 (Product and concatenation laws). *The counting polynomial is*

$$P_n(z) = \sum_s c_n(s) z^s = \prod_{j=1}^n (1 + z^{w_j}). \quad (5)$$

For two blocks of weights with counts $a(s)$ and $b(s)$, concatenation has count $c(t) = \sum_s a(s)b(t - s)$.

Proof. Selecting 1 or z^{w_j} from factor j corresponds to excluding or including position j . Multiplication groups exactly those choices having the same total exponent. Equivalently, multiplying the recurrence by z^s and summing gives $P_i(z) = (1 + z^{w_i})P_{i-1}(z)$. For two blocks, their independent choices form ordered pairs, so multiplicities multiply before equal total sums are combined. This is precisely coefficient convolution. $\square$

Corollary 4 (An exact exploration count). *The reference implementation performs exactly*

$$C_n = 2 \sum_{i=0}^{n-1} W_i \quad (6)$$

candidate contributions when it finishes without a resource-limit exception. Also, if $S_i = \sum_{j=1}^i w_j$, then

$$W_i \leq \min(2^i, S_i + 1). \quad (7)$$

Proof. Each old state is visited once and generates exactly two contributions, including when both contributions have the same destination. Summing over layers proves (6). There are 2^i histories and every reachable sum is an integer between zero and S_i , proving the two width bounds. $\square$

Corollary 5 (Many histories, few states). *For $w_j = 1$ for every j , one has*

$$R_i = \{0, \dots, i\}, \quad c_i(s) = \binom{i}{s}, \quad C_n = n(n+1). \quad (8)$$

Thus 2^n histories have an exact final representation with $n+1$ counts.

Proof. The sum of a binary vector equals its number of selected positions. There are $\binom{i}{s}$ vectors with exactly s ones, and every s between zero and i occurs. Substituting $W_i = i+1$ into (6) gives $2 \sum_{i=0}^{n-1} (i+1) = n(n+1)$. $\square$

The relative representation shrinks compared with the history space; the absolute count $n(n+1)$ still grows. This distinction is essential.

Proposition 6 (A family without state compression). *For $w_j = 2^{j-1}$, every history has a distinct sum. Hence $W_i = 2^i$ and $C_n = 2(2^n - 1)$.*

Proof. Binary representation gives a bijection from length- i bit vectors to $\{0, \dots, 2^i - 1\}$. Substituting its cardinality into (6) and evaluating the geometric sum proves the claim. $\square$

3 SAMPLING THINNING ON A DECLARED RATIONAL GRID

This second model is independent of aggregation. Fix an integer $B \geq 0$. For an integer scale $L \geq 1$, define the closed feasible interval and its integer samples by

$$I_L = [0, B/L], \quad G_L = I_L \cap \mathbb{Z}, \quad K_B(L) = |G_L|. \quad (9)$$

The origin is included. The grid spacing is always one; it is the feasible interval that contracts. The word “scale” names the parameter L , not the binary input length or the number of independent variables.

Theorem 7 (Exact sampling-thinning law). *For the grid in (9),*

$$K_B(L) = \left\lfloor \frac{B}{L} \right\rfloor + 1. \quad (10)$$

For fixed B , this count is nonincreasing in L . For each integer $k \geq 1$,

$$K_B(L) \leq k \iff B < kL. \quad (11)$$

In particular, $K_B(L) = 1$ if and only if $L > B$.

Proof. The grid consists of the consecutive integers $0, 1, \dots, \lfloor B/L \rfloor$, proving (10). If $L' \geq L > 0$, then $B/L' \leq B/L$; the floor function preserves this ordering. To prove (11), use the exact equivalence $\lfloor x \rfloor \leq k - 1$ if and only if $x < k$ for integer k . With $x = B/L$, multiplication by the positive L gives $B < kL$. The grid is nonempty, so the case $k = 1$ gives the final assertion. $\square$

Corollary 8 (Sharp boundary and integer certification). *For $k, L \geq 1$, the instance $B = kL - 1$ has exactly k samples, whereas $B = kL$ has exactly $k + 1$. A proposed count $K \geq 1$ is correct precisely when*

$$(K - 1)L \leq B < KL. \quad (12)$$

Proof. Integer division at the two adjacent boundaries proves the first statement. For the second, divide (12) by $L > 0$. It states exactly that $K - 1$ is the floor of B/L . $\square$

Corollary 9 (Preservation on nested grids). *Let f be any function on G_L , and let $L' \geq L$. If an old maximizer $j^* \in G_L$ belongs to $G_{L'}$, then j^* remains a maximizer on $G_{L'}$, with the same optimal value.*

Proof. The interval endpoints give $G_{L'} \subseteq G_L$. Every point in the smaller set therefore has value at most $f(j^*)$, while j^* remains feasible. $\square$

The retention hypothesis cannot be omitted. For $B = 24$ and $f(j) = 4j - j^2$, the grid $G_1 = \{0, \dots, 24\}$ has unique maximum $f(2) = 4$. But $G_{25} = \{0\}$ has maximum zero. Thinning has changed the optimization task and its answer. Neither the theorem nor the demonstration claims that an arbitrary function’s continuous maximum can be recovered from these samples. A theorem about a declared finite grid is not a theorem about all points of an interval.

4 IMPLEMENTATION, WORK METRICS, AND LIMITS

The companion `pft.py` is an independently written public artifact. It uses built-in integers and standard-library facilities only. Exact integer division implements (10); no floating-point approximation enters a proof check. Python’s integer arithmetic is documented in the language’s standard-types reference [4].

4.1 Reference interfaces

`aggregate_binary(weights)` returns sorted counts, one indexed witness per sum, all layer widths, the candidate-contribution count, and the number of histories. Only exact equality permits coalescence. A configurable state limit raises an exception before a new state is inserted beyond the limit. It never changes the result into a beam-search approximation.

`verify_witness(weights, target, indices)` checks distinct, in-range indices and the exact selected sum. This checks a particular witness, not the completeness of the count distribution. The latter rests on Theorem 2 and the independent tests.

`sampling_count(B,L)` and `sampling_within_budget(B,L,k)` implement (10) and (11). The latter also handles $k = 0$: the answer is false because every grid contains the origin. `integer_grid_maximum` evaluates an integer-coefficient polynomial on every point of the declared grid, using Horner's rule and choosing the smallest index on a tie. Its evaluation count is exactly $K_B(L)$.

4.2 What is and is not counted

Equation (6) counts candidate contributions, not all machine instructions. Dictionary operations, witness-tuple copies, memory allocation, integer arithmetic, input validation, and final sorting add work. In particular, a witness can contain i indices, and multiplicities can require $i + 1$ bits. The implementation stores one such witness per reachable state. Writing $W_{\max} = \max_i W_i$, its stored witness indices can occupy $O(nW_{\max})$ slots, in addition to count and sum integers and the width log. No constant-cost arithmetic model is silently used to claim a wall-time bound.

Similarly, computing a grid count by division differs from evaluating a function at every sample. For a degree- r polynomial, the implemented Horner loop performs $r + 1$ multiply-add updates per sample; operand sizes affect their bit cost. A larger L can have a longer binary encoding even while $K_B(L)$ falls. No CPU-cycle or elapsed-time speedup is reported here.

The bound $W_i \leq S_i + 1$ depends on numeric magnitude and may be exponential in the binary encoding length of the weights. Proposition 6 exhibits exponential width directly. There is no all-input polynomial-time conclusion for arbitrary subset-sum instances.

5 REPRODUCIBLE EXAMPLES AND VALIDATION

All tables below are produced by the public demonstration, using inputs specified in this manuscript. They describe executed operation counts and exact values; no comparison against an unexecuted solver is implied.

Table 1: Unit-weight aggregation. The last column counts candidate contributions.

n	Histories 2^n	Final states $n + 1$	C_n
4	16	5	20
8	256	9	72
12	4096	13	156
16	65536	17	272

Table 2: Declared-grid thinning for $B = 24$ and $f(j) = 4j - j^2$.

L	Endpoint B/L	Samples	Maximizer	Maximum
1	24	25	2	4
2	12	13	2	4
3	8	9	2	4
6	4	5	2	4
12	2	3	2	4
25	24/25	1	0	0

5.1 Independent-oracle test design

The companion `test_pft.py` uses Python's `unittest` framework and executable `doctest` examples [5]. Its exhaustive aggregation oracle enumerates binary vectors directly, rather than repeating the production recurrence. It compares the

full count distribution for all 1365 weight sequences of lengths zero through five over $\{0, 1, 2, 3\}$. Each returned witness is checked by direct indexed summation. Mass conservation and the exact transition count are checked as well.

Further tests cover binomial coefficients through length 40, the distinct-sum family through length 12, zero-weight multiplicities through length 100, 150 seeded permutation/scaling cases, and 100 seeded concatenation cases. The grid oracle enumerates the defining inequality for all 2592 pairs $0 \leq B \leq 80, 1 \leq L \leq 32$. It checks budget decisions on both sides of the returned count. Another 576 cases test the strict threshold at adjacent integers. Polynomial maxima are compared against direct power evaluation in 500 cases. Tests also cover arbitrary-precision division boundaries, invalid types, forged witnesses, exact resource-limit behavior, deterministic output, isolated command-line execution, and propagation of doctest failures.

The recorded Python 3.9.6 run passed all 22 unit tests, 5560 parameterized subcases, and 20 doctest examples across both files. These are distinct reporting units: subcases are not separate test methods. The JSON test receipt records the executed counts, interpreter version, pass status, and SHA-256 hashes of the two Python sources. Test cases are finite evidence for the implementation; they do not substitute for the mathematical proofs. The suite explicitly includes the changed-optimum and exponential-width counterexamples to prevent success cases from defining the claimed scope.

5.2 Reproduction commands

Place the two Python files in the same directory and run:

```
python3 pft.py --self-test
python3 pft.py
python3 test_pft.py --report pft_test_results.json
python3 -m unittest -v test_pft
```

Python 3.9 or later is sufficient. No optional package, credential, private dataset, network endpoint, or external executable is required. Every normal command-line invocation of `pft.py` first calls `doctest.testmod()`. The manuscript compiles independently of the Python files and JSON receipt.

6 DISCUSSION: WHICH SCALING CLAIM SURVIVES?

Three claims should be kept separate. First, an exponentially large history space can have a compact exact representation when the requested output depends on a sufficient state. This is an information-preservation statement. Second, a declared feasible interval can admit fewer grid points as its scale parameter increases. This is a domain-cardinality statement. Third, actual runtime may improve, remain flat, or worsen depending on representation costs, arithmetic precision, and implementation. That is an empirical statement not established by either cardinality formula alone.

The two models suggest useful questions in resource accounting, configuration enumeration, reliability counts, and bounded integer optimization. In each application, a proposed state must preserve the needed future behavior, and a proposed thinning rule must state whether it preserves the original feasible solutions. Applications requiring identity-sensitive histories or preservation of an optimum excluded by the new grid do not satisfy those requirements.

The title's "narrower searches" therefore has two precise meanings: fewer representative states than histories, or fewer samples in a smaller declared region. It does not mean that enlarging every input makes every computation cheaper. These distinctions turn an appealing scaling intuition into a set of falsifiable, mathematically explicit claims.

7 CONCLUSION

Exact aggregation replaces redundant histories by sufficient states while preserving counts and witnesses. Rational-grid thinning admits a sharp integer sample-count law and an exact budget certificate. Their limits are equally explicit: state width can remain exponential, and removing samples can remove the optimizer. The public implementation demonstrates precisely these results, with independent oracles and counterexample tests. A credible rethinking of scaling begins by identifying what changes, what is counted, and what must remain true.

ARTIFACT AND PREPARATION STATEMENT

The public artifacts are `pft.py`, `test_pft.py`, and the machine-readable test receipt. The implementation was written independently for the finite models defined here. It is not an implementation or validation of an undisclosed method. AI assistance was used to draft the exposition, implementation, and test suite. Mathematical claims are accompanied by their proofs and executable checks; author review remains necessary before submission. No journal acceptance, assigned issue, or publication date is asserted.

REFERENCES

- [1] Massachusetts Institute of Technology, “Lecture 18: Dynamic Programming, Part 4: Rods, Subset Sum, Pseudopolynomial,” *6.006 Introduction to Algorithms*, MIT OpenCourseWare, Spring 2020. <https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/lecture-18-dynamic-programming-part-4-rods-subset-sum-pseudopolynomial/>.
- [2] R. P. Stanley, *Enumerative Combinatorics*, vol. 1, 2nd ed., Cambridge University Press, 2012. Author-hosted text: <https://math.mit.edu/~rstan/ec/ec1.pdf>.
- [3] M. S. Yang, “Hilbert–Krylov Tower Decomposition and a Pseudo-Polynomial Complexity Bound for Subset Sum,” *International Journal of Computer Techniques*, vol. 12, no. 6, article IJCT-V12I6P57, 2025. <https://ijctjournal.org/hilbert-krylov-pseudo-polynomial-complexity/>.
- [4] Python Software Foundation, “Built-in Types: Numeric Types,” *Python 3 Documentation*. <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>.
- [5] Python Software Foundation, “unittest—Unit Testing Framework” and “doctest—Test Interactive Python Examples,” *Python 3 Documentation*. <https://docs.python.org/3/library/unittest.html>; <https://docs.python.org/3/library/doctest.html>.

A PUBLIC PYTHON REFERENCE IMPLEMENTATION

The complete source of `pft.py` is reproduced below, including its embedded examples and command-line entry point. Line numbers are editorial; all source characters, indentation, and blank lines are preserved. Keywords are blue, strings red, comments green, numbers purple, and built-ins teal. This listing contains only the independent public models described in the paper.

Source SHA-256:

2f938d599d95d6cb5e91a963d6142a784572a94fd09eba639d454669999e75ae

```

1  #!/usr/bin/env python3
2  """Public, independent demonstrations of exact aggregation and sampling thinning.
3
4  This is not a translation, implementation, or validation of a private algorithm.
5  Only finite integer sums and a declared rational sampling grid are used. Python
6  3.9+ and the standard library suffice. No files or services are read by the demos.
7
8  >>> aggregate_binary([1, 1]).counts
9  ((0, 1), (1, 3), (2, 3), (3, 1))
10 >>> sampling_count(12, 4)
11 4
12 >>> sampling_count(12, 13)
13 1
14 >>> sampling_within_budget(12, 4, 3)
15 False
16 >>> sampling_within_budget(12, 5, 3)
17 True
18 """
19
20 from dataclasses import dataclass
21 from fractions import Fraction
22 import argparse
23 import doctest
24 import json
25 import sys
26
27
28 def _integer(value, name, minimum=0):
29     if type(value) is not int:
30         raise TypeError(name + " must be a built-in integer (not bool)")
31     if value < minimum:
32         raise ValueError(name + " must be >= " + str(minimum))
33     return value
34
35
36 @dataclass(frozen=True)
37 class Aggregation:
38     """Immutable, sorted counts and one index witness per reachable sum.
39
40     widths includes the initial singleton state. transitions counts two
41     candidate contributions per old state; it is not CPU time or bit cost.
42     """
43
44     counts: tuple
45     witnesses: tuple
46     widths: tuple
47     transitions: int
48     histories: int
49
50
51 class StateLimitExceeded(RuntimeError):
52     """Exact aggregation stopped; no approximate result is returned."""
53
54
55 def aggregate_binary(weights, *, max_states=100_000):
56     """Count all binary selections, coalescing only identical partial sums.
```

```

57
58 Zero weights and duplicates denote distinct selectable positions. Negative
59 weights are excluded from this public model. Witnesses use input indices.
60 A resource limit aborts rather than silently pruning reachable states.
61
62 >>> aggregate_binary([]).counts
63 ((0, 1),)
64 >>> aggregate_binary([0, 0]).counts
65 ((0, 4),)
66 >>> aggregate_binary([2, 2]).witnesses
67 ((0, ()), (2, (1,)), (4, (0, 1)))
68 >>> try:
69 ...     aggregate_binary([1, 2], max_states=2)
70 ... except StateLimitExceeded as error:
71 ...     print(str(error))
72 exact state limit exceeded at layer 2
73 """
74 _integer(max_states, "max_states", 1)
75 values = tuple(weights)
76 for weight in values:
77     _integer(weight, "weight")
78 counts = {0: 1}
79 witnesses = {0: ()}
80 widths = [1]
81 transitions = 0
82 for index, weight in enumerate(values):
83     next_counts = {}
84     next_witnesses = {}
85     # Input order and dictionary insertion order give deterministic witnesses.
86     for total, multiplicity in counts.items():
87         for chosen in (False, True):
88             candidate = total + (weight if chosen else 0)
89             transitions += 1
90             if candidate not in next_counts:
91                 if len(next_counts) >= max_states:
92                     raise StateLimitExceeded(
93                         "exact state limit exceeded at layer " + str(index + 1)
94                     )
95                 next_counts[candidate] = 0
96                 next_witnesses[candidate] = (
97                     witnesses[total] + (index,) if chosen else witnesses[total]
98                 )
99                 next_counts[candidate] += multiplicity
100     counts, witnesses = next_counts, next_witnesses
101     widths.append(len(counts))
102 return Aggregation(
103     tuple(sorted(counts.items())), tuple(sorted(witnesses.items())),
104     tuple(widths), transitions, 1 << len(values)
105 )
106
107
108 def verify_witness(weights, target, indices):
109     """Verify an indexed selection without running the aggregation algorithm.
110
111     >>> verify_witness([2, 2, 5], 4, (0, 1))
112     True
113     >>> verify_witness([2, 2, 5], 4, (0, 0))
114     False
115     >>> verify_witness([2, 2, 5], 4, (True, 1))
116     False
117     """
118     try:
119         values = tuple(weights)
120         chosen = tuple(indices)
121         _integer(target, "target")
122         for weight in values:

```

```

123         _integer(weight, "weight")
124         if any(type(i) is not int or i < 0 or i >= len(values) for i in chosen):
125             return False
126         if len(set(chosen)) != len(chosen):
127             return False
128         return sum(values[i] for i in chosen) == target
129     except (TypeError, ValueError):
130         return False
131
132
133 def sampling_count(budget, scale):
134     """Count integer j >= 0 with scale*j <= budget, including j=0.
135
136     This is an exact grid count, not a general optimization/search solver.
137     >>> sampling_count(0, 9)
138     1
139     >>> sampling_count(17, 4)
140     5
141     >>> sampling_count(4, 0)
142     Traceback (most recent call last):
143     ...
144     ValueError: scale must be >= 1
145     """
146     _integer(budget, "budget")
147     _integer(scale, "scale", 1)
148     return budget // scale + 1
149
150
151 def sampling_within_budget(budget, scale, max_samples):
152     """Exact threshold: K <= k iff budget < k*scale, for k >= 1.
153
154     >>> sampling_within_budget(0, 1, 0)
155     False
156     >>> sampling_within_budget(10, 11, 1)
157     True
158     """
159     _integer(budget, "budget")
160     _integer(scale, "scale", 1)
161     _integer(max_samples, "max_samples")
162     return budget < max_samples * scale
163
164
165 def integer_grid_maximum(coefficients, budget, scale, *, max_samples=100_000):
166     """Maximize a polynomial over ONLY the declared finite integer grid.
167
168     Coefficients are ascending-power integers. Exact values use Horner's rule.
169     Ties choose the smallest index. This makes no continuous-domain guarantee.
170
171     >>> integer_grid_maximum([0, 4, -1], 12, 3)
172     (2, 4, 5)
173     >>> integer_grid_maximum([0], 5, 2)
174     (0, 0, 3)
175     """
176     values = tuple(coefficients)
177     if not values or any(type(c) is not int for c in values):
178         raise ValueError("coefficients must be a nonempty sequence of integers")
179     _integer(max_samples, "max_samples", 1)
180     count = sampling_count(budget, scale)
181     if count > max_samples:
182         raise StateLimitExceeded("exact sampling limit exceeded")
183     best_index, best_value = 0, None
184     for index in range(count):
185         value = 0
186         for coefficient in reversed(values):
187             value = value * index + coefficient
188         if best_value is None or value > best_value:

```

```

189         best_index, best_value = index, value
190     return best_index, best_value, count
191
192
193 def demo():
194     """Return deterministic public examples, with no machine-speed claims."""
195     aggregation = []
196     for length in (4, 8, 12, 16):
197         result = aggregate_binary([1] * length)
198         aggregation.append({
199             "length": length, "histories": result.histories,
200             "final_states": len(result.counts), "transitions": result.transitions,
201             "mass_verified": sum(c for _, c in result.counts) == result.histories,
202         })
203     thinning = []
204     for scale in (1, 2, 3, 6, 12, 25):
205         best_index, best_value, samples = integer_grid_maximum([0, 4, -1], 24, scale)
206         thinning.append({
207             "scale": scale, "interval_end": str(Fraction(24, scale)),
208             "samples": samples, "best_index": best_index, "best_value": best_value,
209         })
210     return {
211         "scope": "independent public aggregation and finite-grid demonstrations",
212         "aggregation": aggregation, "thinning": thinning,
213         "caution": "Different scales define different feasible grids; no universal speedup.",
214     }
215
216
217 def main(argv=None):
218     parser = argparse.ArgumentParser(description=__doc__.splitlines()[0])
219     parser.add_argument("--self-test", action="store_true", help="run embedded examples only")
220     args = parser.parse_args(argv)
221     # Explicit testmod() on every CLI execution, as requested for this project.
222     result = doctest.testmod(sys.modules[__name__])
223     if result.failed:
224         return 1
225     if args.self_test:
226         print(json.dumps({"doctests": result.attempted, "failures": result.failed}))
227     else:
228         print(json.dumps(demo(), indent=2))
229     return 0
230
231
232 if __name__ == "__main__":
233     raise SystemExit(main())

```