Segment-Based Dynamic Programming for Optimal Two-Way Comparison Search Trees: Hierarchical Weight Partitioning with Provable Pruning

Anonymous authors
Paper under double-blind review

ABSTRACT

A binary search tree built from the comparisons that programming languages actually provide, "$q < k$" and "$q = k$", can be much cheaper than one built from Knuth's three-way comparisons, but it is also much harder to optimize. After an equality test the remaining queries form a *punctured interval*. The best exact algorithm (Chrobak, Golin, Munro and Young, 2022) therefore runs in $O(n^4)$ time and $\Theta(n^3)$ space, and the Knuth–Yao speed-up does not apply. We present **Hierarchical Weight Partitioning (HWP)**, a segment-based dynamic program with three provable ingredients. (i) A *canonical state space* stores each punctured interval exactly once, which cuts memory from $n^3/2$ to $n^3/6 + O(n^2)$ words and enables a cache-friendly, wavefront-parallel loop order. (ii) A *multi-level segment tree* (merge-sort tree) answers rank and weight queries on punctured intervals in $O(\log^2 n)$. (iii) *Weight-class thresholding* admits an equality test only when the tested key carries at least a fraction τ of the subproblem weight. With dyadic weight classes, we show that at most $\lceil (2-\tau)/\tau \rceil - 1$ keys per class can ever be tested for equality. For successful-query instances this makes HWP with $\tau = 1/4$ exact in $O(n^3 \log 2R)$ time, where R is the key-weight ratio; the previous bound was $O\big(n^3 \log(nR)\big)$. For the general variant, every $\tau \in [0,1]$ yields a tree within additive 3 (in normalized weight) of optimal, $\tau = 0$ is exact, and an equality test on a key heavier than half the subproblem is always optimal (tight). We also prove that no lower threshold can be exact in the general variant. Experiments on synthetic families and four real workloads (English vocabulary with out-of-vocabulary queries, CPython identifiers, opcodes and lexer characters) certify exactness against brute force and the $O(n^4)$ baseline on over 4,000 instances. They also measure scaling up to $n = 3200$, ablations, skew sensitivity and compiled-lookup deployment. We do **not** claim a sub-cubic exact algorithm for the general problem, and we explain why the recurrence rules one out.

Keywords: optimal binary search trees; two-way comparisons; dynamic programming; pruning; weight thresholds; decision procedures; code generation.

1 INTRODUCTION

Static search trees are precomputed decision procedures. Given n keys and the frequency with which each key, and each gap between consecutive keys, is queried, they minimize the expected number of comparisons per lookup. They appear wherever a fixed dictionary is queried many times. Examples are method dispatch in object-oriented runtimes (Chambers & Chen, 1999), the lowering of `switch` statements and interpreter dispatch loops, lexers that classify characters or keywords, static vocabularies in tokenizers, and the comparison-based last mile of learned and classical index structures (Kraska et al., 2018). Knuth (1971) gave the celebrated $O(n^2)$ algorithm for the classical model, in which each node performs a *three-way* comparison of the query q with a key k (outcomes $q < k$, $q = k$, $q > k$). Real instruction sets and programming languages, however, offer *two-way* comparisons ("$q < k$?" or "$q = k$?"), each with two outcomes. Knuth himself posed their optimization as an exercise (Knuth, 1998, §6.2.2, Ex. 33).

Why two-way comparisons are hard. In a three-way tree every node is reached by an open interval between two keys, which yields $O(n^2)$ subproblems. In a two-way tree an equality test removes a single key from the middle of the current interval, so later nodes are reached by *punctured intervals*: an interval minus the keys already tested for equality (Figure 1b). In general, exponentially many punctured intervals can arise.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Anderson, Kannan, Karloff and Ladner (2002) showed that optimal trees need only test the *most likely key* for equality. Chrobak, Golin, Munro and Young (2022, "CGMY") refined this into the RMLK property (Theorem 1 below), which reduces the state space to $O(n^3)$ valid sets and yields a simple $O(n^4)$ algorithm, the fastest known exact method for the general (*full*) variant. The Knuth–Yao speed-up (Knuth, 1971; Yao, 1980; Bein et al., 2009) does not transfer. The quadrangle inequality and root monotonicity both fail for two-way trees, and there are instances on which the CGMY recurrence creates $\Omega(n^3)$ subproblems, each needing $\Omega(n)$ work (Atalig et al., 2023). In practice the binding constraint is often memory. A dense table for the CGMY recurrence with $n = 1000$ keys occupies 4 GB.

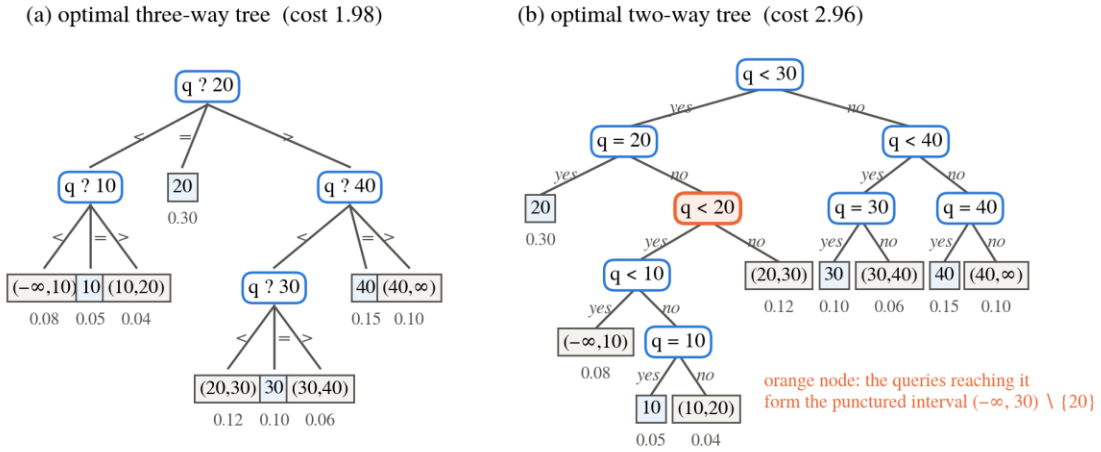

Figure 1: The same instance (keys 10, 20, 30, 40 with key weights β and gap weights α printed under the leaves) solved (a) with three-way comparisons (Knuth, 1971) and (b) with two-way comparisons (this paper; identical to CGMY). In (b) the node "$q < 20$" is reached by the punctured interval $(-\infty, 30) \setminus \{20\}$. Replacing each three-way node of (a) by two two-way tests costs 3.18, which is 7% more than the optimum 2.96 of (b). The three-way cost 1.98 counts one comparison per three-way node, so it is not comparable with (b).

This paper. We revisit the segment-based dynamic-programming idea of partitioning keys by weight and make every component provable. Hierarchical Weight Partitioning (HWP, Figure 2) combines:

- **Canonical states (§4.1).** Every interval (i, j) containing m keys has exactly $m + 1$ distinct punctured versions, which we index by their number of holes t. The resulting exact dynamic program stores $\frac{(n+1)(n^2+8n+6)}{6} = n^3/6 + O(n^2)$ values (Proposition 1), about $3\times$ fewer than the dense CGMY table. It also admits an *interval-blocked* loop order that reads child states contiguously.
- **A multi-level segment tree (§4.2).** A merge-sort tree over key positions answers "heaviest remaining key", "remaining weight" and "number of remaining keys" for any punctured interval in $O(\log^2 n)$ (Lemma 2). It drives the construction of removal chains without scanning intervals.
- **Weight-class thresholding (§4.3–4.4).** An equality test on the heaviest key k of a subproblem S is *admissible* if $\beta_k \geq \tau\, w(S)$. We prove a *dyadic class bound*: among keys whose weights lie in a common class $[2^c, 2^{c+1})$, at most $Q(\tau) = \lceil (2-\tau)/\tau \rceil - 1$ can ever be tested for equality inside one interval (Lemma 3). This bounds the number of states per interval by $O(Q(\tau)\log 2R)$ (Theorem 2). We then characterize when thresholding is exact:
  - for successful-query instances, $\tau = 1/4$ is exact, which gives an $O(n^3 \log 2R)$ algorithm (Theorem 3);
  - for the full variant, an equality test on a key heavier than half the subproblem is always optimal, and the constant $1/2$ is tight (Proposition 4);
  - for the full variant, *no* positive lower threshold is exact (Proposition 5);
  - for the full variant, every $\tau \in [0,1]$ nevertheless returns a tree of cost at most $\mathrm{OPT} + 3W$, with $\tau = 0$ exact (Theorem 6). The threshold therefore acts as a dial from exactness to speed with a worst-case guarantee.

- **Evaluation (§5).** Exactness is certified against exhaustive search and against CGMY on over 4,000 instances. We report scaling with fitted exponents, component ablations, skew sensitivity (Zipf and Pareto weights), four real workloads, heuristic and approximation baselines, and a deployment study in which the trees are compiled into lookup code. Code, data-generation scripts and proofs are released.

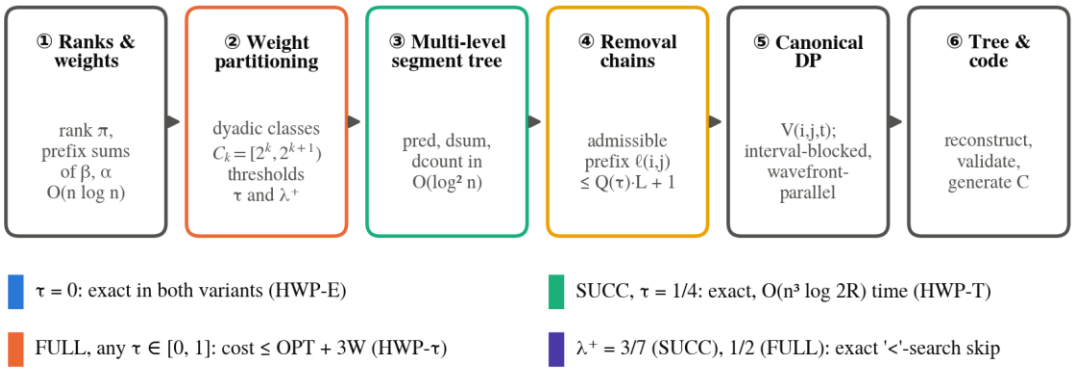

Figure 2: The HWP pipeline and the guarantee attached to each configuration. R is the ratio between the largest and smallest positive key weights, and W is the total weight.

What we do not claim. HWP does not break the cubic barrier for exact two-way search trees in general. Under the CGMY recurrence every exact method must evaluate all $\Theta(n^3)$ canonical states on some inputs, and the split minimization over those states is a $(\min, +)$-type product for which the classical monotonicity tools are unavailable (§4.2, §6). Our exact sub-quartic result holds for the successful-query variant and bounded weight ratios, where it refines the $O(n^3 \log(nR))$ bound of Atalig et al. (2023). For the full variant, HWP is exact at $\tau = 0$ with a $3\times$ smaller memory footprint, and approximate with a proven additive guarantee for $\tau > 0$.

Organization. Section 2 reviews the models and related work, and Table 1 compares HWP with existing methods. Section 3 fixes notation and the CGMY recurrence. Section 4 presents HWP and its guarantees. Section 5 reports experiments, Section 6 discusses generality and limitations, and Section 7 concludes. Proofs are in Appendix A, pseudocode is in Appendix B, and extended results are in the supplementary material.

2 Background and Related Work

Three-way versus two-way comparisons. A *three-way* comparison node "q ? k" has three children, for $q < k$, $q = k$ and $q > k$, and the middle child is the leaf of k (Figure 1a). A *two-way* comparison node asks a single yes/no question, either "$q < k$?" or "$q = k$?", and has two children. Mainstream languages and ISAs implement two-way comparisons, so a three-way node costs two machine comparisons unless the hardware exposes a three-way branch. Two-way trees are strictly more expressive per comparison. For example, they can test a very frequent key first with a single equality test. They are also harder to optimize, because an equality test removes a key from the *interior* of the current interval.

Exact algorithms, three-way model. The standard dynamic program over the $O(n^2)$ inter-key intervals runs in $O(n^3)$ time (Cormen et al., 2009, §15.5). Knuth (1971) proved that optimal roots are monotone, $\mathrm{root}(i, j-1) \le \mathrm{root}(i, j) \le \mathrm{root}(i+1, j)$, which reduces the time to $O(n^2)$. F. F. Yao (1980) generalized this speed-up to every recurrence whose cost satisfies the *quadrangle inequality*, and Bein et al. (2009) showed that the speed-up follows from total monotonicity. We stress that **no $\Omega(n^3)$ lower bound exists for the three-way problem**: its exact complexity is $O(n^2)$. When only gaps carry weight, the problem becomes optimal alphabetic coding, which is solvable in $O(n \log n)$ time (Hu & Tucker, 1971; Garsia & Wachs, 1977).

Exact algorithms, two-way model. Spuler (1994) conjectured the most-likely-key (MLK) property and proposed an unproven $O(n^5)$ algorithm; a second algorithm in that work was later shown to rest on incorrect recurrences (see CGMY, 2022). Anderson et al. (2002) proved the MLK property for the successful-query variant. Via thresholds on the weight of a key that can be tested at the root, they obtained the first

polynomial, $O(n^4)$, algorithm. They showed in particular that if the heaviest key has less than $1/4$ of the total weight, no optimal tree starts with an equality test ($\lambda^- = 1/4$). Chrobak et al. (2015; 2022) gave a simple $O(n^4)$ algorithm that also solves the full variant with unsuccessful queries, based on the refined MLK property (Theorem 1). Atalig and Chrobak (2025) proved the tight upper threshold $\lambda^+ = 3/7$: if the heaviest key has more than $3/7$ of the weight, some optimal tree starts with an equality test. Atalig, Chrobak, Mousavian, Sgall and Veselý (2023) showed that the quadrangle inequality and root monotonicity *fail* for two-way trees, gave instances on which the CGMY recurrence needs $\Omega(n^4)$ time, and obtained an $O\big(n^3 \log(nR)\big)$ algorithm for successful-query instances whose key-weight ratio is R. Chrobak et al. (2021) bounded the extra cost of identifying unsuccessful queries by 1. Chrobak and Young (2025) extended the two-way model to classification, where several keys share a label.

Approximation. Weight-balanced ("nearly optimal") trees approximate the optimum to within entropy-based bounds (Mehlhorn, 1975). For two-way trees, Chrobak et al. (2015) gave an $O(n \log n)$ additive-3 approximation. It merges each key with its right gap, builds an alphabetic tree over the merged items (Yeung, 1991), and appends one equality test per leaf. Its cost is at most $W(H + 3) \leq \mathrm{OPT} + 3W$, because $\mathrm{OPT} \geq W\,H$ for any binary decision tree whose leaves are the $2n + 1$ outcomes.

Search trees versus decision trees in machine learning. Learning an optimal *classification* tree over a multi-attribute feature space is NP-hard for common objectives (Hyafil & Rivest, 1976). In practice it is attacked with dynamic programming, search and pruning (Demirović et al., 2022; van der Linden et al., 2023; Brița et al., 2025; Harviainen et al., 2025). The problem studied here is different and polynomial. There is a *single* totally ordered attribute, every leaf is a known outcome, and the objective is the expected number of comparisons. We therefore do not claim that HWP learns classifiers. The methodological link is optimality-preserving state pruning in dynamic programs, as in separator-based pruning for Steiner trees (Iwata & Shigemura, 2019). Our applications are the ones where two-way decision procedures over one ordered key are actually deployed: dispatch, static dictionaries, lexers and switch lowering (§5.8).

Table 1: Exact and approximate methods for optimal search trees. "Full" means that unsuccessful queries (gaps) are supported. W = total weight, H = entropy, R = key-weight ratio, $Q(\tau) = \lceil (2 - \tau)/\tau \rceil - 1$.

Method	Model / variant	Guarantee	Time	Space
Knuth (1971); Yao (1980)	three-way, full	exact	$O(n^2)$	$O(n^2)$
Hu–Tucker; Garsia–Wachs	"<" only (alphabetic)	exact	$O(n \log n)$	$O(n)$
Mehlhorn (1975)	three-way	entropy-type bound	$O(n \log n)$	$O(n)$
Spuler (1994)	two-way, succ.	unproven	$O(n^5)$	$O(n^3)$
Anderson et al. (2002)	two-way, succ.	exact	$O(n^4)$	$O(n^3)$
CGMY (2015; 2022)	two-way, full	exact	$O(n^4)$	$\approx n^3/2$ words
CGMY (2015), additive-3	two-way, full	$\leq \mathrm{OPT} + 3W$	$O(n \log n)$	$O(n)$
Atalig et al. (2023)	two-way, succ.	exact	$O\big(n^3 \log(nR)\big)$	$O\big(n^2 \log(nR)\big)$
HWP-E (this work)	two-way, full & succ.	exact	$n^4/12 + O(n^3)$ splits	$n^3/6 + O(n^2)$ words
HWP-T, $\tau = \frac{1}{4}$ (this work)	two-way, succ.	exact	$O(n^3 \log 2R)$	$O(n^2 \log 2R)$
HWP-τ (this work)	two-way, full	$\leq \mathrm{OPT} + 3W$	$O(Q\, n^3 \log 2R)$, $Q = Q(\tau)$	$O(Q\, n^2 \log 2R)$

3 Problem Formulation and the CGMY Recurrence

Instances. Let $k_1 < k_2 < \cdots < k_n$ be the keys and set $k_0 = -\infty$, $k_{n+1} = +\infty$. A query q is either one of the keys or a non-key value, and every non-key value lies in exactly one *gap* $g_a = (k_a, k_{a+1})$, $a = 0, \ldots, n$. Each key k_b has a weight $\beta_b \geq 0$ (the frequency of the query $q = k_b$). Each gap g_a has a weight $\alpha_a \geq 0$ (the frequency of queries that fall strictly between k_a and k_{a+1}). For example, α_0 is the weight of queries below k_1 and α_n that of queries above k_n. We call keys and gaps *elements* and write $w(x)$ for the weight of element x and $W = \sum_b \beta_b + \sum_a \alpha_a$ for the total weight. Weights need not be normalized.

Two-way comparison trees. A two-way comparison tree T is a rooted binary tree. Every internal node N carries a test "$q < k_b$?" or "$q = k_b$?" for some key k_b and has a *yes* child and a *no* child. Every leaf carries one element. A query is routed from the root by the outcomes of the tests, and T is *correct* if every query reaches the leaf of its own element. In the **full** variant (FULL) there are $2n + 1$ leaves, one per key and one per gap. In the **successful-query** variant (SUCC) only keys are queried, the α_a are irrelevant and there are n leaves. Chrobak et al. (2022) show that SUCC is not a special case of FULL. With $\mathrm{depth}_T(x)$ the number of edges from the root to the leaf of x, and w_N the total weight of the elements whose leaves lie below node N,

$$\mathrm{cost}(T) = \sum_x w(x)\,\mathrm{depth}_T(x) = \sum_{N \text{ internal}} w_N\,. \qquad (1)$$

The goal is a correct tree of minimum cost, OPT. The second form of (1) holds because every internal node on the path to x contributes $w(x)$ once.

Why trees have right children (reviewer question). Every internal node has two children. After "$q < k_b$?" the *yes* child handles the queries below k_b and the *no* child those at or above k_b. After "$q = k_b$?" the *yes* child is the leaf of k_b and the *no* child handles everything else. The most-likely-key property below constrains *which* key an equality node may test. It says nothing about "<" nodes, and it does not force trees to be one-sided (Figure 1b).

Ranks and valid sets. Order the keys by weight: let π be the permutation with $\beta_{\pi(1)} \leq \cdots \leq \beta_{\pi(n)}$, where ties are broken by index, and let $\mathrm{rank}(b) = \pi^{-1}(b)$. The key of largest rank in a set is therefore its heaviest key, with the largest index among equal weights. For $0 \leq i < j \leq n + 1$ and $0 \leq h \leq n$ define the **valid set**

$$S(i, j, h) = \{g_a : i \leq a < j\} \cup \{k_b : \max(i, 1) \leq b < j,\ \mathrm{rank}(b) \leq h\}. \qquad (2)$$

This is the stretch of the line from k_i (inclusive) to k_j (exclusive), from which all keys of rank greater than h have been removed; those removed keys are the *holes*. A larger h means fewer holes: $S(0, n + 1, n)$ is the whole line and $S(i, j, 0)$ contains only gaps. Let $w(S)$ denote the total weight of S, and $\mathrm{opt}(i, j, h)$ the minimum cost of a tree that correctly handles exactly the queries in $S(i, j, h)$. In SUCC we drop the gaps from (2) and use $1 \leq i$. We write $K(i, j) = \{b : \max(i, 1) \leq b \leq j - 1\}$ for the key positions of an interval and $m_{ij} = |K(i, j)|$.

Theorem 1 (refined most-likely-key property; Chrobak et al., 2022, Thm. 1). *Let S be a valid set with at least two elements and let k be any key of largest weight in S. Some optimal tree for S either (i) has a "<" test at its root, or (ii) has the test "$q = k$" at its root.*

Applying Theorem 1 with $k = k_{\pi(h)}$, the key of largest rank in $S(i, j, h)$, gives the **CGMY recurrence**. Its three cases are labeled (a)–(c) and are referenced throughout.

- **(a)** If $j = i + 1$ and $k_i \notin S(i, j, h)$, the set is the single gap g_i and $\mathrm{opt}(i, j, h) = 0$.
- **(b)** If $h > 0$ and $k_{\pi(h)} \notin S(i, j, h)$, the key of rank h lies outside the interval, $S(i, j, h) = S(i, j, h - 1)$, and $\mathrm{opt}(i, j, h) = \mathrm{opt}(i, j, h - 1)$.
- **(c)** Otherwise

$$\mathrm{opt}(i, j, h) = w\big(S(i, j, h)\big) + \min\Big\{\mathrm{opt}(i, j, h - 1),\ \min_{i<b<j}\big[\mathrm{opt}(i, b, h) + \mathrm{opt}(b, j, h)\big]\Big\}. \qquad (3)$$

The first term of the minimum is the equality test "$q = k_{\pi(h)}$" (present only if $h > 0$). Its *yes* child is the leaf of $k_{\pi(h)}$, and its *no* child handles $S(i, j, h) \setminus \{k_{\pi(h)}\} = S(i, j, h - 1)$. The second term is the inequality test "$q < k_b$", where b ranges over the key indices strictly inside the interval. That test splits $S(i, j, h)$ into its part below k_b, which is $S(i, b, h)$, and its part at or above k_b, which is $S(b, j, h)$. The answer is $\mathrm{opt}(0, n +$

$1, n)$. There are $O(n^3)$ states (i, j, h) and each case-(c) state costs $O(n)$, so the algorithm runs in $O(n^4)$ time. Table 6 in Appendix B collects all notation.

4 HIERARCHICAL WEIGHT PARTITIONING

HWP keeps the CGMY recurrence as its semantic core and changes three things: how states are represented (§4.1), how the weight of a state and its heaviest key are found (§4.2), and which equality tests are explored (§4.3–4.4). Figure 2 summarizes the pipeline.

4.1 CANONICAL STATES AND THE INTERVAL-BLOCKED RECURRENCE

For a fixed interval (i, j), the set $S(i, j, h)$ depends on h only through *which* keys of $K(i, j)$ have rank above h. Since ranks are distinct, these are always the heaviest keys of the interval. Let $r_1 > r_2 > \cdots > r_m$ be the ranks of the $m = m_{ij}$ keys of $K(i, j)$, set $r_{m+1} = 0$, and for $t \in \{0, \ldots, m\}$ let $S_t(i, j) = S(i, j, r_{t+1})$ be the interval with its t heaviest keys removed. We call (i, j, t) a **canonical state**, write $W_t = w\big(S_t(i, j)\big)$, and let $V(i, j, t)$ denote its value.

Proposition 1 (canonical states). *(a) For every h, $S(i, j, h) = S_{t(h)}(i, j)$ with $t(h) = |\{b \in K(i, j) : \mathrm{rank}(b) > h\}|$. Hence $\mathrm{opt}(i, j, h) = V\big(i, j, t(h)\big)$, and an interval has exactly $m_{ij} + 1$ distinct valid sets. (b) The number of canonical states is*

$$N_{\mathrm{F}}(n) = \frac{(n+1)(n^2+8n+6)}{6} \text{ (FULL)}, \qquad N_{\mathrm{S}}(n) = \frac{n(n+1)(n+5)}{6} \text{ (SUCC)}, \tag{4}$$

whereas a dense $(i < j,\ h)$ table for the CGMY recurrence has $(n+1)^2(n+2)/2$ cells (FULL). (c) The values satisfy $V(i, j, t) = 0$ if $S_t(i, j)$ is a single element or $W_t = 0$. Otherwise

$$V(i, j, t) = W_t + \min\{E_t,\ L_t\}, \quad E_t = V(i, j, t+1)\ [t < m], \quad L_t = \min_{i<b<j} \big[V\big(i, b, t_L(b)\big) + V\big(b, j, t - t_L(b)\big)\big], \tag{5}$$

where $t_L(b) = |\{c \in K(i, b) : \mathrm{rank}(c) > r_{t+1}\}|$ is the number of holes to the left of k_b and $E_t = +\infty$ if $t = m$. (d) Evaluating (5) for all states takes exactly $\frac{n(n+1)(n+2)}{3} + \frac{(n-1)n(n+1)(n+4)}{12} = \frac{n^4}{12} + O(n^3)$ split evaluations in FULL, the same number as the case-(c) evaluations of the CGMY recurrence.

Part (b) gives the memory reduction ($n^3/2 \to n^3/6$ words, about 3 ×). Part (d) states the hidden constant of the exact method explicitly. Equation (5) is (3) rewritten in canonical coordinates: an equality test moves from t to $t + 1$, and an inequality test at k_b distributes the t holes between the two sides. Children are again canonical because the holes falling into $K(i, b)$ are exactly the heaviest keys of $K(i, b)$ (Appendix A.1).

Interval-blocked loop order. The split term L_t does not depend on $V(i, j, \cdot)$. HWP therefore computes L_t for all t of an interval together, with b in the outer loop and t in the inner loop. For a fixed b, the index $t_L(b)$ is non-decreasing in t: it increases exactly when the key removed at step t lies left of k_b. Both child blocks $V(i, b, \cdot)$ and $V(b, j, \cdot)$ are thus read *sequentially*. The equality chain $V(i, j, \ell), \ldots, V(i, j, 0)$ is then resolved in one backward pass. Each interval's states occupy a contiguous block (row-major in (i, j)), so the left children of an interval are also adjacent in memory (Figure 10b, Appendix B). When a chain is short ($\ell(i, j) < 4$, the common case under thresholding), the bookkeeping of the blocked order outweighs its benefit. HWP then switches, per interval, to the plain state-outer loop, which gives an *adaptive* loop order ablated in §5.5. Intervals of equal length $j - i$ are independent. HWP therefore runs as a wavefront over $d = j - i$ with $n + 1$ barriers, which is the basis of the parallel implementation and of the communication model in §5.9.

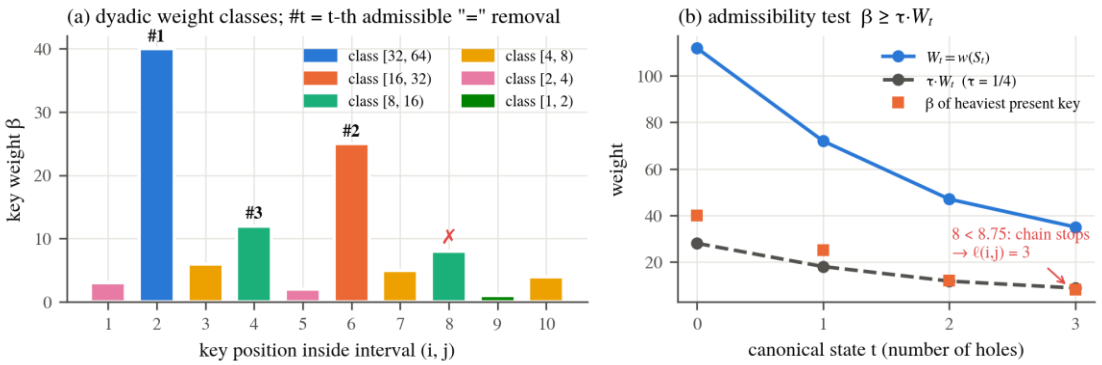

Figure 3: Weight partitioning inside one interval ($\tau = 1/4$). (a) Keys are colored by dyadic weight class. Labels #1–#3 mark the admissible equality removals in heaviest-first order. (b) The chain stops at the first key whose weight falls below τW_t (here $8 < 8.75$), so this interval has only $\ell = 3$ extra states instead of $m = 10$.

4.2 The multi-level segment tree

HWP needs three queries on a canonical state, all of them on an axis-parallel box "key positions $lo..hi$, ranks $\leq h$": the heaviest remaining key (**pred**, the largest rank $\leq h$), the remaining key weight (**dsum**) and the number of remaining keys (**dcount**). We answer them with a *multi-level segment tree*, better known as a merge-sort tree. It is a segment tree over key positions in which every node stores the ranks of its keys in decreasing order, together with prefix sums of the corresponding weights (Figure 9, Appendix B).

Lemma 2. *The multi-level segment tree uses $O(n\log n)$ space and build time and answers pred, dsum and dcount in $O(\log^2 n)$ time (or $O(\log n)$ with fractional cascading). Consequently the removal chain of an interval, meaning the thresholds $r_1, \dots, r_{\ell+1}$ and the weights $W_0, \dots, W_\ell$, is computed in $O\big((\ell+1)\log^2 n\big)$ time without scanning the interval.*

What the tree cannot do. It is tempting to also answer the split minimum in (3) or (5) with a range-minimum structure over b. This does not work, because the summand $\mathrm{opt}(i,b,h) + \mathrm{opt}(b,j,h)$ depends on *both* endpoints. A structure indexed by b would have to be rebuilt for every pair (i,j), which costs $\Omega(j-i)$, the same as the direct loop. Over all pairs and a fixed h, the computation is a $(\min,+)$ product of two triangular matrices. Sub-cubic evaluation of such products needs extra structure, such as the quadrangle inequality or total monotonicity (Yao, 1980; Bein et al., 2009), and that structure *fails* for two-way trees (Atalig et al., 2023). For general matrices no truly sub-cubic algorithm is known (Williams, 2018). HWP therefore saves work by *reducing the number of states* (§4.3) and never by claiming a cheaper split minimum.

4.3 Weight-class thresholding

Admissibility and chains. Fix $\tau \in [0,1]$. At canonical state t of (i,j), let $p = m - t \geq 1$ be the number of remaining keys, $W_t > 0$, and let k be the heaviest remaining key (rank r_{t+1}). The equality test "$q = k$" is **τ-admissible** if

$$\beta_k \;\geq\; \tau\, W_t, \qquad (6)$$

or if $p = 1$ in FULL. The last remaining key must be separated from its right gap, and only an equality test can do that. The **chain length** $\ell(i,j)$ is the number of leading admissible removals $r_1, r_2, \dots$. The chain also stops at the first t with $W_t = 0$, because $V = 0$ from then on. HWP-τ evaluates (5) only for $t \leq \ell(i,j)$, sets $E_t = +\infty$ at $t = \ell(i,j)$, and clamps child indices to the child's chain length. Clamping is exact, because a child chain can be shorter than the parent's hole count only through a zero-weight tail (Appendix A.2).

Let $\mathcal{F}_\tau$ be the set of correct trees in which every equality node tests the heaviest key present at that node, ties being broken by rank, and is τ-admissible. Admissibility is inherited by subproblems: a key admissible in a set remains admissible in every subset that contains it, because the set weight only decreases. Hence $\mathcal{F}_\tau$ is closed under taking subtrees, and HWP-τ computes $\min_{T \in \mathcal{F}_\tau} \mathrm{cost}(T)$ exactly (Appendix A.2). Since $\mathcal{F}_{\tau'} \subseteq \mathcal{F}_\tau$ for $\tau \leq \tau'$, the returned cost is non-decreasing in τ. By Theorem 1, $\mathcal{F}_0$ contains an optimal tree, so **HWP-E** (= HWP-0) is exact in both variants.

Hierarchical weight partitioning. Partition the keys into dyadic weight classes $C_c = \{b : \beta_b \in [2^c, 2^{c+1})\}$, and let L_{ij} be the number of classes that meet $K(i,j)$. Figure 3 illustrates the next lemma, which is the formal content of the intuition that only a few heavy keys per class need special treatment.

Lemma 3 (dyadic class bound). *Let $\tau \in (0,1]$. If an interval contains $q_c \geq (2-\tau)/\tau$ keys of class C_c, then no key of C_c is removed by an admissible equality test with $W_t > 0$ in its chain, except possibly the FULL last-key test. Consequently the chain removes at most $Q(\tau) = \lceil (2-\tau)/\tau \rceil - 1$ keys of any class, and $\ell(i,j) \leq \min\{m_{ij},\ Q(\tau)\, L_{ij} + 1\}$.*

Proof sketch. Keys are removed in decreasing weight order. When the first key x of C_c is removed, all q_c keys of C_c are therefore still present, so $W_t \geq \beta_x + (q_c - 1)2^c$. Admissibility (6) gives $\tau(\beta_x + (q_c - 1)2^c) \leq \beta_x < 2^{c+1}$. Rearranging yields $q_c < (2-\tau)/\tau$. ▫

For $\tau = 1/4$ we get $Q = 6$; for $\tau = 1/2$, $Q = 2$; and for $\tau = 1$, $Q = 0$. Since $L_{ij} \leq \lfloor \log_2 R \rfloor + 1 \leq \log_2(2R)$, where R is the ratio between the largest and smallest positive key weight:

Theorem 2 (complexity of HWP-τ). *For $\tau \in (0,1]$, HWP-τ stores at most $\sum_{i<j} (\ell(i,j) + 1) = O(Q(\tau)\, n^2 \log 2R)$ values. It performs at most $\sum_{i<j} (\ell(i,j) + 1)\,(j - i - 1) = O(Q(\tau)\, n^3 \log 2R)$ split evaluations, plus $O(Q(\tau)\, n^2 \log(2R) \log^2 n)$ time for chain construction with the multi-level segment tree.*

4.4 When is thresholding exact?

Theorem 3 (successful queries: exact sub-quartic HWP). *In the SUCC variant, (a) HWP-τ with $\tau = 1/4$ returns an optimal tree, and (b) whenever the heaviest remaining key satisfies $\beta_k > \frac{3}{7} W_t$, the split minimization L_t can be skipped ($V = W_t + E_t$) without loss. Hence **HWP-T** (τ = 1/4 with the λ^+ = 3/7 rule) computes* OPT *in $O(n^3 \log 2R)$ time and $O(n^2 \log 2R)$ space.*

Part (a) follows from the threshold of Anderson et al. (2002): if $\beta_k < \frac{1}{4} w(S)$, *no* optimal tree for S starts with an equality test. Every inadmissible state therefore has an optimal "<" root, and $\mathcal{F}_{1/4}$ contains an optimal tree. Part (b) combines $\lambda^+ = 3/7$ (Atalig & Chrobak, 2025) with the fact that, for successful queries, an optimal equality root can use *any* maximum-weight key (Chrobak et al., 2022, full version, App. B). The running time follows from Theorem 2 with $Q(1/4) = 6$. Compared with the $O\big(n^3 \log(nR)\big)$ bound of Atalig et al. (2023), the dyadic class bound removes the $\log n$ factor. For bounded R it gives $O(n^3)$ directly, and for Zipf-like weights with $R = n^s$ both bounds are $O(n^3 \log n)$.

Proposition 4 (full variant: tight upper threshold $1/2$). *In FULL, if a valid set S contains a key k with $\beta_k > w(S)/2$, then every optimal tree for S starts with "$q = k$". The same holds for the minimum over $\mathcal{F}_\tau$ whenever that test is τ-admissible. The constant is tight: for every $c < 1/2$ there is an instance with $\beta_k > c\,W$ in which no optimal tree starts with an equality test.*

Proof sketch. In FULL the leaf of k must hang below "$q = k$", since only that test separates k from its right gap. If the root is not that node, move "$q = k$" to the root. Every node on the former path loses weight β_k, and the former parent node disappears. The cost changes by $W - w_P - r\beta_k \leq W - 2\beta_k < 0$, where $r \geq 1$ is the depth of the former node P testing k. For tightness take $n = 1$, $\beta_1 = c' \in \left(c, \frac{1}{2}\right)$, $\alpha_0 = 1 - c'$, $\alpha_1 = 0$: starting with "$q < k_1$" costs $1 + c' < 2 - c'$. ▫

Proposition 5 (full variant: no exact lower threshold). *For every $c > 0$ there is a FULL instance with a single key of weight $\beta < c\,W$ whose unique optimal tree starts with an equality test. Thus the $\lambda^- = 1/4$ rule has no exact analogue in FULL.*

Proof. Take $n = 1$, $\beta_1 = \varepsilon$, $\alpha_0 = \varepsilon/2$, $\alpha_1 = 1$. Testing "$q = k_1$" first costs $W + \alpha_0 + \alpha_1$. Testing "$q < k_1$" first costs $W + \beta_1 + \alpha_1$, which is larger by $\varepsilon/2$, while $\beta_1/W \to 0$. ▫

Propositions 4 and 5 explain why thresholding is exact for successful queries but cannot be exact for the full variant. Gap weight can make an early equality test worthwhile even for a light key, because it lets the next inequality test isolate a heavy gap. HWP nevertheless keeps a worst-case guarantee for every threshold:

Theorem 6 (full variant: additive guarantee for every τ). *For every FULL instance and every $\tau \in [0,1]$, HWP-τ, with or without the $\lambda^+ = 1/2$ rule of Proposition 4, returns a correct tree T_τ with*

$$\mathrm{OPT} \le \mathrm{cost}(T_\tau) \le \mathrm{OPT} + 3W. \qquad (7)$$

Moreover $\mathrm{cost}(T_\tau)$ *is non-decreasing in* τ*, and* T_0 *is optimal.*

Proof sketch. Let $\mathcal{A}$ be the family used by the additive-3 approximation of Chrobak et al. (2015): an alphabetic "<"-tree over the items $\{g_0, k_1, g_1\}, \{k_2, g_2\}, \dots, \{k_n, g_n\}$, completed by equality tests at its leaves. Every equality node of such a tree acts on a set with a single key, so it is admissible for every τ. Hence $\mathcal{A} \subseteq \mathcal{F}_\tau \subseteq \mathcal{F}_0$, and $\min_{\mathcal{F}_\tau} \le \min_{\mathcal{A}} \le W(H+3) \le \mathrm{OPT} + 3W$. The last step uses the entropy lower bound $\mathrm{OPT} \ge WH$. The λ^+ rule does not change $\min_{\mathcal{F}_\tau}$, by the exchange argument of Proposition 4 carried out inside $\mathcal{F}_\tau$. ▫

Proposition 7 (thresholding never hurts the worst case). *The canonical states evaluated by HWP-*τ *map injectively to case-(c) states of the CGMY recurrence, and each performs at most the same split loop. For every* τ*, HWP-*τ *performs at most as many split evaluations as CGMY (Proposition 1d), and chain construction adds* $O(n^3 \log n)$ *time in the worst case.*

Proposition 7 answers the question of whether the adaptive step can worsen the worst-case running time: it cannot. On the hard family of Atalig et al. (2023) chains are long and HWP-T degrades gracefully to the exact $O(n^4)$ behavior (§5.3).

Algorithm. The complete procedure (Algorithm 1, Appendix B) has two passes. Pass 1 computes all chain lengths with the multi-level segment tree and allocates one contiguous block per interval. Pass 2 evaluates (5) wavefront by wavefront. Trees are reconstructed from stored choices and validated by routing all $2n + 1$ queries.

5 Experiments

The experiments answer six questions. Are the exact configurations exact (§5.2)? How do time and memory scale, and up to which n can exact trees be built (§5.3)? How good are thresholded trees compared with existing heuristics and approximations (§5.4)? What does each component contribute (§5.5)? How do skewed, power-law weights affect thresholding (§5.6)? Does any of this matter on real workloads and in deployed lookup code (§5.7–5.8)? §5.9 covers parallel execution.

5.1 Setup

Implementations. All methods share one C++17 code base (`g++ 13.3 -O3 -march=native`), with one process per run. *CGMY* is the published recurrence with a dense, h-major table. *CGMY-vec* is our fastest CGMY implementation, with an h-contiguous layout and a SIMD split loop over all h; it performs twice the useful work at vector speed. *HWP-E* is HWP with $\tau = 0$ (exact). *HWP-T* is $\tau = 1/4$ with $\lambda^+ = 3/7$ (exact, SUCC). *HWP-*τ is the thresholded method with $\lambda^+ = 1/2$ (FULL). The baselines are:

- *Knuth3→2W*: the optimal three-way tree, with each node replaced by the locally cheaper pair of two-way tests;
- *Balanced*: a frequency-oblivious binary-search tree;
- *A3 (GW)*: the additive-3 approximation of Chrobak et al. (2015), realized with Garsia–Wachs;
- *WB-greedy*: a weight-balanced greedy tree that tests the heaviest key for equality when it carries at least 40% of the weight and otherwise splits at the weighted median;
- *Huffman LB*: an order-free lower bound.

Data. There are seven synthetic weight families: uniform, equal, Zipf with $s \in \{0.5,1,1.5,2\}$, Pareto(1.1), clustered, and the *hard* geometric family of Atalig et al. (2023). In FULL instances gaps carry 20% of the weight. There are four real workloads: **opcodes** (CPython 3.11 bytecode, SUCC, $n = 97$); **punct** (ASCII punctuation as lexer classes, all other characters as unsuccessful queries, FULL, $n = 32$); **words**N (the N most frequent English words from wordfreq; the next $3N$ words are out-of-vocabulary queries mapped to gaps; FULL, $N \le 2048$); and **ident**N (the most frequent identifiers of the CPython standard library, SUCC, $N \le 4000$). Supplementary S3 documents their construction.

Protocol. One machine: 2 vCPUs (Intel Xeon, 2.1 GHz), 8 GB RAM (about 7 GB usable). Runs are single-threaded unless stated. We report wall-clock time, peak resident memory (VmHWM) and the objective itself, as cost/W or the gap to OPT. Values are mean ± sd over 5 seeds (3 seeds for $n \le 400$, 1 seed for larger n).

Hardware counters and energy were not accessible. Cache behavior is simulated with Cachegrind (Supp. S5.8).

5.2 Exactness is certified

Table 2: Correctness certification (Supp. S4). "Violations" counts any disagreement with the stated guarantee.

Check	Instances	Comparisons	Violations
CGMY and HWP-E vs. exhaustive search, $n \le 6$ (FULL 1,500; SUCC 1,441)	2,941	10,382	0
HWP-T ($\tau = 1/4$, with/without $\lambda^+ = 3/7$) vs. exhaustive search, SUCC	1,441	4,323	0
HWP-τ within $[\mathrm{OPT}, \mathrm{OPT} + 3W]$, $\tau \in \{0.1, 0.25, 0.5, 1\}$, FULL	1,500	6,000	0
All implementations cross-checked, $n \le 120$, 9 families, both variants	144	1,152	0
Garsia–Wachs vs. $O(m^3)$ alphabetic DP	20,000	20,000	0
Exact runs of §5.3–5.7 agreeing with another exact method	145	453	0

Every tree reported in the paper is reconstructed and validated by routing all $2n + 1$ queries. The DP value must equal the recomputed tree cost.

5.3 Scaling: time, memory and the largest exact instances

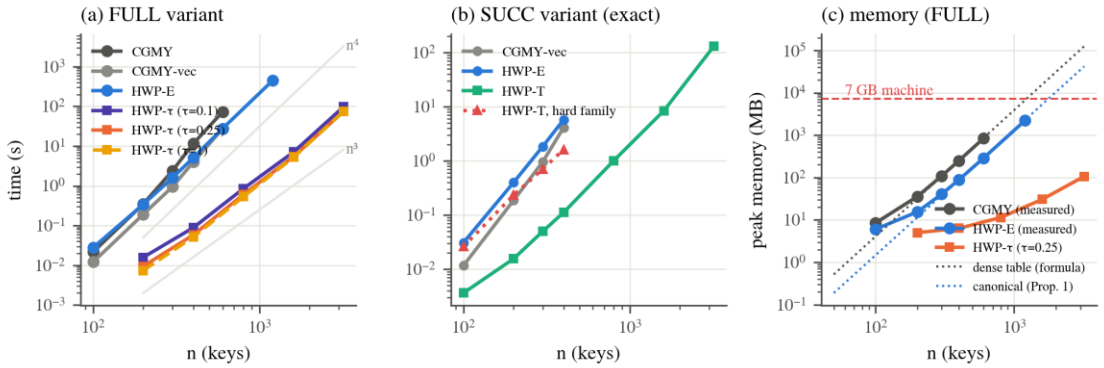

Figure 6: Scaling on log–log axes, mean over families and seeds. (a) FULL: exact methods (CGMY, CGMY-vec, HWP-E) and thresholded HWP-τ. Grey guides show n^4 and n^3 slopes. (b) SUCC: exact CGMY-vec, HWP-E and HWP-T; dotted red is HWP-T on the hard family of Atalig et al. (2023). (c) Peak memory (FULL) with the closed forms of Proposition 1 and the 7 GB limit of the machine.

Figure 6 and Table 8 (Appendix C) summarize scaling. HWP-E follows its $\Theta(n^4)$ work closely (fitted exponent 4.03 for $n \ge 200$). The dense CGMY tables show larger apparent exponents (4.90 and 4.55) because their working sets outgrow the caches. HWP-E performs exactly the predicted $n^4/12 + O(n^3)$ split evaluations at 2.39 ns each. It is 2.6× faster than the dense CGMY table and within 1.02× of the vectorized CGMY-vec, while using 2.9× less memory (Proposition 1). Memory decides the largest exact instance on this machine. CGMY's table needs 4.0 GB at $n = 1000$ and cannot be allocated beyond $n \approx 1050$. HWP-E solves $n = 1200$ exactly in 7.5 min using 2.2 GB, where the dense table would need 6.9 GB.

Thresholding changes the exponent. HWP-τ grows as $n^{3.42}$ ($\tau = 0.25$) and $n^{3.48}$ ($\tau = 1$), consistent with the $O(n^3 \log 2R)$ bound of Theorem 2 (here $R \approx n$). It builds trees with $n = 3200$ keys in 74.6 s and 106 MB, with a measured gap to the optimum of at most 0.0002% wherever the optimum is available. In the SUCC variant HWP-T is *exact*. It scales as $n^{3.29}$, reaching $n = 3200$ in 2.2 min. On the hard family, however, chains are long (mean $\ell \approx 30$), and HWP-T is no faster than HWP-E, exactly as Proposition 7 and the lower bound of Atalig et al. predict.

5.4 Quality against existing methods

Table 3 compares all methods at $n = 400$, where the exact optimum is available for every instance (6 families × 5 seeds). The thresholded trees are essentially optimal. HWP-τ with $\tau = 0.25$ has a mean gap of 0.0006% and a worst case of 0.0054%, and it runs 62× faster than HWP-E. With $\tau = 0.1$ the worst gap is 0.0013%. The measured additive excess stays far below the guaranteed $3W$: at most $1.18W$ even at $\tau = 1$. At $\tau = 1$ HWP-τ coincides with the A3 (GW) tree, which Theorem 6 predicts because both optimize over the alphabetic family $\mathcal{A}$. Among the fast heuristics, the weight-balanced greedy tree is strong (mean gap 0.74%) but has no guarantee. The three-way conversion loses 20.4%, and the frequency-oblivious tree loses 43.7% on average. At larger scale, where the optimum is unknown, the certified entropy gap $\text{cost}/W - H$ of the heuristics stays below 0.69 bits up to $n = 131{,}072$ (Supp. S5.6).

Table 3: Relative gap to the optimum at $n = 400$ (FULL: 30 instances; SUCC: 30 instances). The fourth column is the largest additive excess in units of W; Theorem 6 guarantees at most 3 for HWP-τ and A3.

Method	Guarantee	FULL: mean gap	FULL: max gap	FULL: max (cost−OPT)/W	FULL: exact / runs	FULL: time	SUCC: mean gap	SUCC: time
HWP-E	exact	0.000%	0.000%	0.0000	30/30	4.9 s	0.000%	5.0 s
HWP-τ (τ=0.05)	≤ OPT+3W	0.000%	0.000%	0.0000	27/30	213 ms	–	–
HWP-τ (τ=0.1)	≤ OPT+3W	0.000%	0.001%	0.0001	24/30	130 ms	–	–
HWP-τ (τ=0.25)	≤ OPT+3W	0.001%	0.005%	0.0002	17/30	79 ms	–	–
HWP-τ (τ=0.5)	≤ OPT+3W	1.902%	5.988%	0.3218	0/30	57 ms	–	–
HWP-τ (τ=1)	≤ OPT+3W	10.1%	26.6%	1.1812	0/30	53 ms	–	–
WB-greedy	none	0.744%	1.487%	0.1276	0/30	284 µs	1.122%	310 µs
A3 (GW)	≤ OPT+3W	10.1%	26.6%	1.1812	0/30	119 µs	5.990%	85 µs
Knuth3→2W	none	20.4%	26.5%	2.2885	0/30	1 ms	20.8%	1 ms
Balanced	none	43.7%	123.6%	5.4876	0/30	51 µs	81.6%	31 µs
Huffman LB	lower bound	-2.053%	-0.463%	-0.0368	0/30	130 µs	-0.180%	63 µs
HWP-T	exact (SUCC)	–	–	–	–	–	0.000%	326 ms

Three-way bucketing (reviewer question). Rounding all weights down to powers of two and solving the three-way problem exactly costs at most 1.07% more than the optimum in our runs, well inside the factor-2 guarantee (§6). The order-free Huffman bound lies below every search tree (Table 9, Appendix C).

5.5 Ablation: what each component contributes

Table 4: Ablation (FULL, $n = 400$, Zipf(1), 5 seeds; SUCC and all variants in Table 10, Appendix C). "Adaptive" is the default loop order (blocked for chains with $\ell \geq 4$, state-outer otherwise).

Configuration	time (mean ± sd)	peak memory	split evaluations	stored states
CGMY	11.711 ± 0.340 s	251 MB	2.18e+09	3.23e+07
CGMY-vec	3.895 ± 0.217 s	251 MB	4.31e+09	3.23e+07
HWP-E [adaptive]	4.798 ± 0.072 s	89 MB	2.18e+09	1.09e+07
HWP-E [blocked]	4.809 ± 0.245 s	89 MB	2.18e+09	1.09e+07

Configuration	time (mean ± sd)	peak memory	split evaluations	stored states
HWP-E [t-outer]	20.034 ± 0.845 s	89 MB	2.18e+09	1.09e+07
HWP-E [blocked, mirror]	4.861 ± 0.191 s	173 MB	2.18e+09	1.09e+07
HWP-E [adaptive, λ^+]	4.733 ± 0.108 s	89 MB	2.18e+09	1.09e+07
HWP-E [adaptive, 2 threads]	2.785 ± 0.641 s	89 MB	2.18e+09	1.09e+07
HWP-τ (τ=0.25) [adaptive]	0.068 ± 0.014 s	6 MB	1.22e+07	1.05e+05
HWP-τ (τ=0.25) [adaptive, chain: sort]	0.769 ± 0.052 s	6 MB	1.22e+07	1.05e+05
HWP-τ (τ=0.25) [blocked]	0.088 ± 0.004 s	6 MB	1.22e+07	1.05e+05
HWP-τ (τ=0.25) [adaptive, λ^+]	0.061 ± 0.005 s	6 MB	1.21e+07	1.05e+05

Table 4 isolates each component.

- **Canonical states** cut memory from 251 MB to 89 MB at identical split counts.
- **Loop order** is the largest constant-factor effect for exact HWP: the state-outer order is 4.2× slower than the adaptive, interval-blocked order. In simulation it incurs about 7× fewer L1-D misses per split than the dense CGMY table (Supp. S5.8). A column-major mirror of the table doubles memory without speeding up the blocked order.
- **The multi-level segment tree** makes chain construction cheap. Building chains by sorting each interval's ranks instead is 11.3× slower for HWP-τ.
- **Thresholding** reduces stored states by 104× and split evaluations by 178× ($\tau = 0.25$).
- **The λ^+ rule** is exact but removes only a few percent of split evaluations on these weights, and it is fully time-neutral.
- **The naive shortcut** of evaluating only ranks at which the interval weight halves and reusing neighboring values is **unsound**. It returned values *below* the optimum on all 340 random instances, with a mean relative error of 45% (Supp. S8). This motivated the admissibility-based thresholds of §4.3.

5.6 Skewed and Power-Law Weights

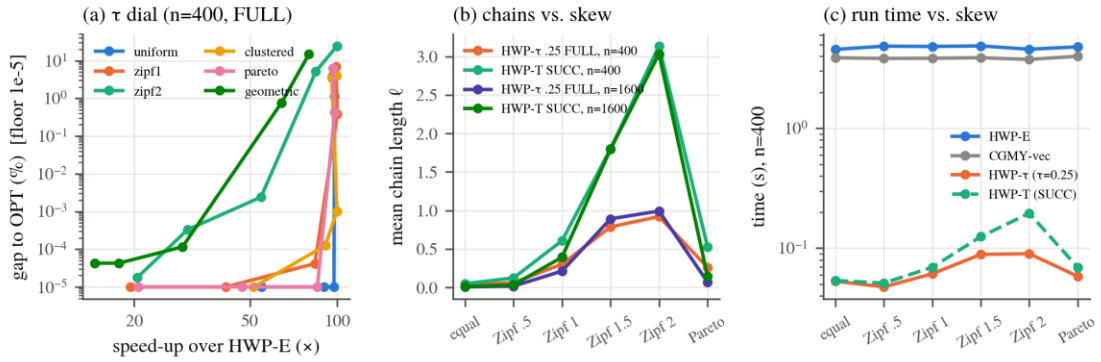

Figure 7: (a) The τ dial at $n = 400$ (FULL). Each curve is one weight family and the points are $\tau \in \{0.05, 0.1, 0.25, 0.5, 1\}$ from left to right; the x-axis is speed-up over HWP-E and the y-axis is the gap to the optimum (log scale, floored at 10^{-5}%). (b) Mean chain length versus weight skew for HWP-τ (FULL) and HWP-T (SUCC) at $n = 400$ and 1600. (c) Run time versus skew at $n = 400$.

Heavier tails create more keys that dominate their intervals. Chains therefore lengthen with skew, but they stay short: the mean ℓ is at most 3.12 even under Zipf(2) (Figure 7b). The run time of HWP-τ and HWP-T varies by less than 1.9× across the skew range, while the exact methods are insensitive to skew by construction (Figure 7c; full numbers in Supp. S5.4). The dial in Figure 7a is well behaved on every family. $\tau \leq 0.25$ keeps the gap below 10^{-2}% at 20–70× speed-ups, and the gap grows only for $\tau \geq 0.5$. The hard geometric family is the one case where thresholding cannot help, because all its keys are heavy relative to their intervals.

5.7 Real workloads

Table 5 reports the real workloads. On the English vocabulary with out-of-vocabulary queries (words1024, FULL), HWP-E needs 3.8 min and 1388 MB. HWP-τ returns a tree of *identical* cost in 1.3 s using 15 MB, 171× faster with 90× less memory. For identifiers (ident1000, SUCC) HWP-T is exact and 165× faster than HWP-E. It also solves ident2000 and ident4000 exactly, where HWP-E would need 85 GB. The best fast heuristic (WB-greedy) is 0.4–1.5% worse than optimal on these instances. Knuth's three-way tree converted to two-way tests is 10–19% worse, and a frequency-oblivious tree is 19–81% worse.

Table 5: Real workloads (var.: F = FULL, S = SUCC; K3→2W = Knuth's three-way tree converted to two-way tests; Bal. = balanced tree). OPT/W is the optimal expected number of two-way comparisons per query. Baseline columns give the gap to the optimum. For FULL instances beyond the reach of HWP-E, the listed value is the HWP-τ upper bound.

Dataset	var.	n	OPT/W	HWP-E (exact): time / mem	HWP-τ / HWP-T: time / mem	gap	WB	A3	K3→2W	Bal.
opcodes	S	97	4.763	23 ms / 6 MB	4 ms / 5 MB	exact	1.5%	3.9%	9.6%	37.3%
punct	F	32	3.309	544 µs / 5 MB	225 µs / 5 MB	0.000%	11.4%	13.2%	9.5%	81.3%
words256	F	256	7.553	965 ms / 27 MB	17 ms / 5 MB	0.002%	0.7%	7.9%	17.6%	19.2%
words512	F	512	8.209	13.6 s / 180 MB	114 ms / 7 MB	0.000%	0.8%	8.2%	19.0%	21.8%
words1024	F	1024	8.861	227 s / 1388 MB	1.3 s / 15 MB	0.000%	0.8%	7.8%	19.1%	24.1%
words2048	F	2048	≤9.443	needs ≈12 GB	12.7 s / 47 MB	n/a	–	–	–	–
ident500	S	500	7.115	12.3 s / 167 MB	155 ms / 7 MB	exact	0.4%	1.1%	17.3%	26.2%
ident1000	S	1000	7.681	206 s / 1290 MB	1.2 s / 16 MB	exact	0.5%	1.5%	17.4%	29.9%
ident2000	S	2000	8.215	needs ≈11 GB	12.2 s / 48 MB	exact	0.6%	1.3%	17.2%	33.7%
ident4000	S	4000	8.743	needs ≈86 GB	202 s / 172 MB	exact	0.7%	1.2%	17.7%	36.9%

5.8 Deployment: compiled lookup code

The optimal tree is computed offline. Every exact method, whether CGMY or HWP, returns a tree of the same cost, so HWP accelerates *construction*, not lookup. The deployment question is whether cost-optimal two-way trees beat the alternatives an engineer would otherwise use. We compiled each tree into flat C code (one `if`/`goto` per node) and timed it on the real query streams (Figure 8):

- the **dynamic** opcode trace of a CPython workload, for a tree built from *static* counts;
- the stdlib source characters in file order;
- 5 M sampled English words including out-of-vocabulary words;
- the identifier token stream of the standard library.



Figure 8: Deployment: median time per lookup over 7 repetitions, compiled with `g++ -O2` (lower is better). Blue = optimal two-way tree (HWP), orange = HWP-τ, green = heuristic trees, dark grey = binary search and hash tables, light grey = `switch` and lookup tables. The two right-hand scenarios use string keys compared with `strcmp`.

Four findings stand out.

1. **The cost model is predictive.** The measured number of comparisons per lookup equals the model cost for in-distribution streams (punct: 3.309 measured vs. 3.309 predicted). On the dynamic opcode trace, a tree built from *static* counts executes 5.12 comparisons (model 4.76), a realistic train/test shift.
2. **Compiled trees beat binary search over a sorted array**, by 4.2× on opcodes and 1.75× on identifiers. Among trees, the optimal one executes the fewest comparisons. The wall-clock gains over a balanced tree are modest for integer keys (1.07× on opcodes), because branch prediction absorbs part of the difference, and larger for strings (1.23× on identifiers).
3. **When the universe is small and dense, a table beats every tree.** A jump table (`switch`) or a 256-entry lookup table takes 1.40 ns versus 3.09 ns for the optimal tree on opcodes. For hashable exact-match string lookups, a hash table is fastest (28.8 ns on identifiers), even with a fallback search to identify the gap of out-of-vocabulary words (67 ns vs. 110 ns).
4. **`strcmp` is a three-way primitive.** The compiler reuses one `strcmp` result for the "=" and "<" tests on the same key. The converted three-way tree is therefore the fastest comparison tree on strings (86 ns), although it executes more two-way tests.

In summary, optimal *two-way* trees are the right tool when the comparison primitive is genuinely two-way (integer, SIMD or hardware decision procedures), when non-key queries must be classified into gaps, and when hashing or dense tables are unavailable. For three-way primitives such as `strcmp`, Knuth's model is the appropriate one. We report these outcomes because they delimit where the construction speed-ups of HWP translate into deployed benefit.

5.9 Parallel execution

Intervals of equal length are independent. With 2 threads, HWP-E runs 1.97× faster at $n = 400$ and HWP-τ 1.53× faster. The wavefront needs $n + 1$ barriers and exposes up to $n + 1$-way parallelism. A BSP model calibrated with the measured split cost predicts near-linear speed-ups for exact HWP-E up to 128 processors: 58× at $P = 64$ for $n = 1500$. Thresholded HWP-τ does little work per interval and becomes communication-bound on a 10 Gb/s network (48× at $P = 64$, $n = 6400$). The model, its assumptions and the memory limit of replication are in Supplementary S6. We do not report multi-node measurements.

6 Discussion: Generality, Other Cost Models, and Limitations

Does HWP generalize to other dynamic programs? Table 7 (Appendix C) separates the components by what they require. *Canonical states* apply whenever DP states are "an interval minus the items that a global order removes first". This covers the valid sets of binary split trees (Huang & Wong, 1984; Hester et al., 1986) and of classification with two-way comparisons (Chrobak & Young, 2025), where it yields the same ≈

$3\times$ memory reduction. The *multi-level segment tree* applies to any DP that queries weights or ranks on (range × threshold) boxes. The *interval-blocked loop order* and the *wavefront schedule* apply to every interval DP, from Knuth-type recurrences to matrix-chain and CYK-style parsing. *Weight-class thresholding* transfers only when a structural threshold theorem is available, such as λ^- for successful two-way queries. The dyadic class bound (Lemma 3) is then generic: any admissibility rule of the form "item weight $\geq \tau \times$ subproblem weight" bounds the chain by $Q(\tau)$ items per class. The *additive dial* (Theorem 6) transfers whenever the restricted family contains a structured family with an entropy guarantee, as alphabetic trees do for comparison trees. What does **not** transfer, and does not hold even here, is sub-cubic split minimization without monotonicity. The bivariate minimum is a $(\min, +)$ product (§4.2).

k-way comparisons and other cost models (reviewer question). The recurrence is unchanged in the richer operator set $\{<, \leq, =\}$: Chrobak et al. (2022) obtain four interval types. Canonical states and the class bound apply per interval type, but the exactness thresholds would need re-derivation. For k-way (multiway) search trees the three-way DP extends (Vaishnavi et al., 1980), and punctured intervals arise only if equality branches are separate. For comparisons between *linear functions* of the inputs, the right framework is A. C.-C. Yao's (1975) linear-comparison decision-tree model, where lower bounds, not constructions, are the concern. Non-uniform comparison costs are handled by replacing w_N in (1) with $c_{\mathrm{op}}\, w_N$. The DP and Theorem 6 carry over with $3\max_{\mathrm{op}} c_{\mathrm{op}} W$, while the thresholds $\lambda^{\pm}$ change.

On power-of-two bucketing and Huffman coding (reviewer question, three-way model). Rounding every weight down to a power of two and solving the rounded instance *exactly* loses at most a factor of 2. For any tree, $\mathrm{cost}_{\tilde{\beta}} \leq \mathrm{cost}_{\beta} < 2\mathrm{cost}_{\tilde{\beta}}$, so the optimum for $\tilde{\beta}$ costs less than $2\,\mathrm{OPT}$ under β. In practice the loss is small (Table 9, Appendix C). Bucketing does not, however, reduce the running time of Knuth's already quadratic algorithm. Huffman coding ignores the key order and therefore does not produce a search tree. Its cost is only a lower bound for any comparison tree, and the order-preserving counterparts are the Hu–Tucker and Garsia–Wachs algorithms. Note also that the three-way problem has no $\Omega(n^3)$ lower bound: it is solvable in $O(n^2)$.

Limitations.

- **Worst case.** For the full variant, the exact method remains $\Theta(n^4)$ in the worst case and $\Theta(n^3)$ in memory. On the family of Atalig et al. (2023) chains are long and HWP-T gives no asymptotic gain. Whether a sub-cubic exact algorithm exists for either variant is open.
- **Full variant.** The thresholded method is approximate there. The worst-case guarantee is additive ($3W$), although the measured gaps are orders of magnitude smaller (§5.4).
- **Hardware and measurement.** All measurements come from one 2-core machine. Cache behavior is reported via simulation (Cachegrind), and energy counters were not accessible. Multi-node results are model-based (§5.9).
- **Static trees.** The trees are static. Dynamic updates would require recomputation, although HWP-τ is fast enough to rebuild trees with thousands of keys in seconds.

7 Conclusion

We revisited segment-based dynamic programming for optimal two-way comparison search trees and gave every component a precise meaning and a proof. Canonical states make the exact algorithm $3\times$ leaner in memory with an explicit $n^4/12$ constant. A multi-level segment tree navigates punctured intervals in polylogarithmic time. Dyadic weight classes turn the classical weight thresholds into a quantitative bound on the number of states. The result is an exact $O(n^3 \log 2R)$ algorithm for successful queries, a tight equality threshold of $1/2$ for the full variant, a proof that no lower threshold can be exact there, and an anytime method with an additive-3 guarantee for every threshold. Experiments on synthetic and real workloads confirm exactness where claimed and quantify the speed/quality trade-off. Compiled lookup code built from these trees shows where optimal two-way trees matter in deployed systems. Closing the gap between $O(n^4)$ and the trivial lower bound for the full variant remains the central open problem.

Reproducibility Statement

All algorithms are implemented in a single C++17 code base, released with scripts that regenerate every table and figure: `run_experiments.py`, `make_figures.py` and the deployment harness. Instances are either generated from documented seeds or derived from public data (wordfreq 3.1; the CPython 3.11 standard library) by the included `make_datasets.py`. Correctness tests compare all exact methods with exhaustive search (`test_correctness.py`) and with each other (`test_large.py`). Every reported tree is re-validated by routing all $2n + 1$ queries. Proofs of all claims are in Appendix A. Pseudocode, extended tables and the dataset construction are in the supplementary material.

Ethics Statement

This work concerns deterministic data-structure construction and uses no personal data. The real-world instances are aggregate token and character frequencies from open-source code and a public word-frequency list. We see no direct risk of harm.

LLM Usage Statement

[To be completed by the authors as required by the ICLR policy on the use of large language models. The revision was prepared with the help of an LLM-based assistant for literature search, code implementation and testing, and editing. All claims, proofs and numbers were checked against the released code and the cited sources.]

References

Anderson, R., Kannan, S., Karloff, H., & Ladner, R. E. (2002). Thresholds and optimal binary comparison search trees. *Journal of Algorithms*, 44(2), 338–358. https://doi.org/10.1016/S0196-6774(02)00203-1

Atalig, S., & Chrobak, M. (2025). A tight threshold bound for search trees with 2-way comparisons. *Theoretical Computer Science*, 1058, 115589. https://doi.org/10.1016/j.tcs.2025.115589 (Conference version: TAMC 2024, LNCS, pp. 99–110.)

Atalig, S., Chrobak, M., Mousavian, E., Sgall, J., & Veselý, P. (2023). Structural properties of search trees with 2-way comparisons. arXiv:2311.02224. https://arxiv.org/abs/2311.02224

Bein, W., Golin, M. J., Larmore, L. L., & Zhang, Y. (2009). The Knuth–Yao quadrangle-inequality speedup is a consequence of total monotonicity. *ACM Transactions on Algorithms*, 6(1), 1–22. https://doi.org/10.1145/1644015.1644032

Brița, C. E., van der Linden, J. G. M., & Demirović, E. (2025). Optimal classification trees for continuous feature data using dynamic programming with branch-and-bound. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(11), 11131–11139. https://doi.org/10.1609/aaai.v39i11.33210

Chambers, C., & Chen, W. (1999). Efficient multiple and predicated dispatching. In *Proceedings of OOPSLA '99* (pp. 238–255). ACM. https://doi.org/10.1145/320384.320407

Chrobak, M., Golin, M., Munro, J. I., & Young, N. E. (2015). Optimal search trees with 2-way comparisons. In *Proceedings of ISAAC 2015*, LNCS 9472 (pp. 71–82). Springer. https://doi.org/10.1007/978-3-662-48971-0_7 (Erratum and full version: arXiv:1505.00357.)

Chrobak, M., Golin, M., Munro, J. I., & Young, N. E. (2021). On the cost of unsuccessful searches in search trees with two-way comparisons. *Information and Computation*, 281, 104707. https://doi.org/10.1016/j.ic.2021.104707

Chrobak, M., Golin, M., Munro, J. I., & Young, N. E. (2022). A simple algorithm for optimal search trees with two-way comparisons. *ACM Transactions on Algorithms*, 18(1), Article 2, 1–11. https://doi.org/10.1145/3477910 (Full version: arXiv:2103.01084.)

Chrobak, M., & Young, N. E. (2025). Classification via two-way comparisons. *ACM Transactions on Algorithms*, 21(2). https://doi.org/10.1145/3709361

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Demirović, E., Lukina, A., Hebrard, E., Chan, J., Bailey, J., Leckie, C., Ramamohanarao, K., & Stuckey, P. J. (2022). MurTree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 23(26), 1–47.

Garsia, A. M., & Wachs, M. L. (1977). A new algorithm for minimum cost binary trees. *SIAM Journal on Computing*, 6(4), 622–642. https://doi.org/10.1137/0206045

Harviainen, J., Sommer, F., Sorge, M., & Szeider, S. (2025). Optimal decision tree pruning revisited: Algorithms and complexity. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*. https://openreview.net/forum?id=iUsHLRTp3t

Hester, J. H., Hirschberg, D. S., Huang, S.-H., & Wong, C. K. (1986). Faster construction of optimal binary split trees. *Journal of Algorithms*, 7(3), 412–424. https://doi.org/10.1016/0196-6774(86)90031-3

Hu, T. C., & Tucker, A. C. (1971). Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4), 514–532. https://doi.org/10.1137/0121057

Huang, S.-H., & Wong, C. K. (1984). Generalized binary split trees. *Acta Informatica*, 21(1), 113–123. https://doi.org/10.1007/BF00289143

Hyafil, L., & Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1), 15–17. https://doi.org/10.1016/0020-0190(76)90095-8

Iwata, Y., & Shigemura, T. (2019). Separator-based pruned dynamic programming for Steiner tree. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01), 1520–1527. https://doi.org/10.1609/aaai.v33i01.33011520

Knuth, D. E. (1971). Optimum binary search trees. *Acta Informatica*, 1(1), 14–25. https://doi.org/10.1007/BF00264289

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018). The case for learned index structures. In *Proceedings of SIGMOD 2018* (pp. 489–504). https://doi.org/10.1145/3183713.3196909

Mehlhorn, K. (1975). Nearly optimal binary search trees. *Acta Informatica*, 5(4), 287–295. https://doi.org/10.1007/BF00264563

Speer, R. (2022). *rspeer/wordfreq: v3.0 (v3.0.2)* [Software]. Zenodo. https://doi.org/10.5281/zenodo.7199437

Spuler, D. (1994). Optimal search trees using two-way key comparisons. *Acta Informatica*, 31(8), 729–740. https://doi.org/10.1007/BF01178732

Vaishnavi, V. K., Kriegel, H. P., & Wood, D. (1980). Optimum multiway search trees. *Acta Informatica*, 14, 119–133. https://doi.org/10.1007/BF00288540

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8), 103–111. https://doi.org/10.1145/79173.79181

van der Linden, J. G. M., de Weerdt, M. M., & Demirović, E. (2023). Necessary and sufficient conditions for optimal decision trees using dynamic programming. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*.

Williams, R. R. (2018). Faster all-pairs shortest paths via circuit complexity. *SIAM Journal on Computing*, 47(5), 1965–1985. https://doi.org/10.1137/15M1024524

Yao, A. C.-C. (1975). On the complexity of comparison problems using linear functions. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (SFCS 1975)* (pp. 85–89). IEEE. https://doi.org/10.1109/SFCS.1975.20

Yao, F. F. (1980). Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC '80)* (pp. 429–435). https://doi.org/10.1145/800141.804691

Yeung, R. W. (1991). Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37(3), 564–572. https://doi.org/10.1109/18.79913

APPENDIX A: PROOFS

Throughout, a tree is *reduced* if no internal node has a child that receives no query. Removing a node with an empty child never increases cost, so optimal trees and the minimizers over $\mathcal{F}_\tau$ may be assumed reduced. In a reduced tree the *yes* child of "$q = k$" is the leaf of k.

A.1 Canonical states (Proposition 1)

(a) By (2), $S(i,j,h)$ contains all gaps $g_i, \dots, g_{j-1}$ and exactly the keys k_b with $b \in K(i,j)$ and $\mathrm{rank}(b) \le h$. Ranks are distinct, so the removed keys, those with $\mathrm{rank}(b) > h$, are the $t(h)$ keys of largest rank in $K(i,j)$. The set therefore equals $S_{t(h)}(i,j)$. As h ranges over $0..n$, $t(h)$ takes every value in $0..m_{ij}$, which gives exactly $m_{ij}+1$ distinct sets.

(b) FULL, $i=0$: $j = 1..n+1$ and $m_{0j} = j-1$, contributing $\sum_{j=1}^{n+1} j = \frac{(n+1)(n+2)}{2}$. For $i \ge 1$: $m_{ij} = j-i =: d$, contributing $\sum_{i=1}^{n}\sum_{d=1}^{n+1-i}(d+1) = \sum_{k=1}^{n}\frac{k(k+3)}{2} = \frac{n(n+1)(n+5)}{6}$. Adding both gives $N_{\mathrm{F}}(n) = \frac{(n+1)(n^2+8n+6)}{6}$. SUCC has only the $i \ge 1$ part, so $N_{\mathrm{S}}(n) = \frac{n(n+1)(n+5)}{6}$. The dense table stores $n+1$ cells for each of the $\binom{n+2}{2}$ intervals, i.e. $(n+1)^2(n+2)/2$.

(c) If $W_t = 0$, every tree for S_t has cost 0 by (1). Otherwise Theorem 1 applies to $S_t(i,j)$, whose heaviest key has rank r_{t+1}. The equality branch leaves $S_t \setminus \{k_{\pi(r_{t+1})}\} = S_{t+1}(i,j)$. For the inequality branch at k_b, the part below k_b is $S(i,b,r_{t+1})$. By (a) applied to the interval (i,b) it equals $S_{t'}(i,b)$ with $t' = |\{c \in K(i,b): \mathrm{rank}(c) > r_{t+1}\}| = t_L(b)$. The part at or above k_b is $S_{t-t_L(b)}(b,j)$, because $K(i,b)$ and $K(b,j)$ partition $K(i,j)$ (note $b \ge 1$).

(d) Every state with $j-i \ge 2$ loops over $j-i-1$ split points. For $i=0$: $\sum_{j=1}^{n+1} j\,(j-1) = \frac{n(n+1)(n+2)}{3}$. For $i \ge 1$: $\sum_{k=1}^{n}\sum_{d=1}^{k}(d+1)\,(d-1) = \frac{(n-1)n(n+1)(n+4)}{12}$. In the CGMY recurrence, case (c) occurs for $h=0$ and for each h equal to the rank of a key in $K(i,j)$, that is, for $m_{ij}+1$ values of h per interval, each with the same loop. States with zero weight skip the loop in both methods, so the counts are equal and are upper bounds in general. ▫

A.2 Correctness of HWP-T and clamping

Closure. Let x be a key removed by an admissible equality test at state t of (i,j), and let $(i',j') \subseteq (i,j)$ be a sub-interval containing x. At the moment x is removed in the chain of (i',j'), the remaining set is $S' \subseteq S_t(i,j)$, because it contains fewer keys and gaps. Hence $w(S') \le W_t$ and $\beta_x \ge \tau W_t \ge \tau\, w(S')$, so x is admissible there too. The FULL last-key exception is inherited as well: if x is the only key left in $S_t(i,j)$, it is the only key left in S'. Therefore every child index $t_L(b)$ (respectively $t - t_L(b)$) arising from a state $t \le \ell(i,j)$ is at most the chain length of the child, *except* when the child's chain stopped early because its weight reached zero. In that case all further removals leave weight zero and value $0 = V(\cdot,\cdot,\ell)$, so clamping to the chain length returns the correct value.

Optimality over $\mathcal{F}_\tau$. By induction on $(j-i,\ m_{ij}-t)$, $V_\tau(i,j,t) = \min\{\mathrm{cost}(T): T \in \mathcal{F}_\tau \text{ handles } S_t(i,j)\}$. The root of such a tree is either an admissible equality test on the heaviest key or an inequality test at some k_b. Its subtrees are again $\mathcal{F}_\tau$-trees of the canonical child sets (closure), and (5) minimizes over exactly these options. When the λ^+ rule is used, Proposition 4 shows that an $\mathcal{F}_\tau$-optimal tree with an equality root exists, so skipping L_t loses nothing. ▫

A.3 Dyadic class bound (Lemma 3) and complexity (Theorem 2)

Let x be the first key of class C_c removed with $W_t > 0$ and not by the last-key exception. Removal is in decreasing rank order, hence in non-increasing weight order. All keys heavier than 2^{c+1} are therefore gone, and all q_c keys of C_c in the interval are still present. So $W_t \ge \beta_x + (q_c-1)2^c$. Admissibility gives $\beta_x \ge \tau W_t \ge \tau\beta_x + \tau(q_c-1)2^c$, i.e. $(q_c-1)2^c \le \beta_x(1-\tau)/\tau < 2^{c+1}(1-\tau)/\tau$. Hence $q_c < (2-\tau)/\tau$. If $q_c \ge (2-\tau)/\tau$ no key of C_c is removed. Otherwise at most $q_c \le \lceil (2-\tau)/\tau \rceil - 1 = Q(\tau)$ are removed. Zero-weight keys can only be removed once $W_t = 0$, which ends the chain, or by the exception. Summing over the L_{ij} classes gives $\ell(i,j) \le Q(\tau)L_{ij} + 1$.

For Theorem 2, each state costs one pass over $j-i-1$ split points plus $O(1)$. The chain of (i,j) costs $O\big((\ell+1)\log^2 n\big)$ by Lemma 2 and is computed twice (once per pass). Summing $\ell(i,j)+1 \le Q(\tau)\log_2(2R)+2$ over the $\binom{n+2}{2}$ intervals, weighted by $j-i-1$ for the split loops, gives the stated bounds. ▫

A.4 Successful queries (Theorem 3)

Let S be a canonical set with heaviest key k and $p = \beta_k / w(S)$. **(a)** If $p < 1/4$, Anderson et al. (2002; restated as Thm. 3.1 in Atalig et al., 2023) show that *no* optimal tree for S has an equality root. Since S is itself a successful-query instance on its remaining keys, every optimal tree starts with "<". If $p \geq 1/4$, Theorem 1 provides an optimal tree whose root is "<" or "$q = k$", and both are allowed in $\mathcal{F}_{1/4}$. Replacing subtrees by optimal $\mathcal{F}_{1/4}$ subtrees (induction) yields an optimal tree in $\mathcal{F}_{1/4}$. **(b)** If $p > 3/7$, some optimal tree has an equality root (Atalig & Chrobak, 2025). For successful queries, an equality root can then be placed on *any* maximum-weight key without loss (Chrobak et al., 2022, full version, App. B), in particular on k, so $V = W_t + V(i, j, t+1)$. The complexity follows from Theorem 2 with $Q(1/4) = 6$. ▫

A.5 Full variant (Propositions 4–5, Theorem 6)

Proposition 4. Let T be a reduced optimal tree (or reduced $\mathcal{F}_\tau$-optimal tree) for S whose root is not "$q = k$", where $\beta_k > w(S)/2$. In FULL, k and its right gap g_k react identically to every "<" test and to every equality test on another key. They are separated only by "$q = k$", whose *yes* child is the leaf of k. Let P be that node, at depth $r \geq 1$, and let $N_0, \ldots, N_{r-1}$ be the nodes above it. None of these tests another key for equality: such a key would be the heaviest present, but k is present with more than half the weight. Build T': root "$q = k$" with *yes* child the leaf of k and *no* child the tree T with P replaced by its *no* child. Then T' is correct. Each N_s loses weight β_k, the node P (weight $w_P \geq \beta_k$) disappears, and the new root has weight $w(S)$. By (1), $\mathrm{cost}(T') - \mathrm{cost}(T) = w(S) - w_P - r\beta_k \leq w(S) - 2\beta_k < 0$, which is a contradiction. If $T \in \mathcal{F}_\tau$ and "$q = k$" is τ-admissible, then $T' \in \mathcal{F}_\tau$, because weights only decrease below the root. Tightness was shown in §4.4.

Proposition 5. Computed in §4.4. The two trees for $n = 1$ are the only reduced trees.

Theorem 6. The family $\mathcal{A}$ consists of trees formed by an alphabetic "<"-tree over the items $I_1 = \{g_0, k_1, g_1\}$ and $I_b = \{k_b, g_b\}$ for $b \geq 2$, completed by "$q = k_b$" at the leaf of I_b and by one "<" and one "=" test in either order at I_1. Every equality node of an $\mathcal{A}$-tree receives a set containing a single key, so it satisfies the last-key exception. It also tests the heaviest key present, so $\mathcal{A} \subseteq \mathcal{F}_\tau$ for all $\tau \in [0,1]$. Every "<" split of an alphabetic tree separates whole items, so the corresponding sets are canonical states with no holes and the DP explores them. Chrobak et al. (2015, Thm. 2) exhibit a tree in $\mathcal{A}$ of cost at most $W(H' + 3)$, where $H' \leq H$ is the entropy of the merged item weights. It is built from Yeung's (1991) alphabetic code, whose cost is at most $W(H' + 2)$, plus at most one extra level per item and one more for I_1. For any binary tree whose leaves are the $2n + 1$ elements, $\mathrm{cost} \geq WH$ (Kraft–Shannon). Hence $\min_{\mathcal{F}_\tau} \leq W(H + 3) \leq \mathrm{OPT} + 3W$. Monotonicity in τ follows from $\mathcal{F}_\tau \subseteq \mathcal{F}_\tau$ for $\tau \leq \tau'$, and exactness at $\tau = 0$ from Theorem 1. The λ^+ rule does not change $\min_{\mathcal{F}_\tau}$ (Proposition 4 inside $\mathcal{F}_\tau$). ▫

A.6 Structural lemmas used by Theorem 1 (corrected statements)

For completeness we restate, with correct proofs following Anderson et al. (2002) and Chrobak et al. (2022, App. A), the two structural lemmas on which Theorem 1 rests. The *side weight* $sw(N)$ of a node is 0 for a leaf, β_k for "$q = k$", and $\min(w_{\mathrm{yes}}, w_{\mathrm{no}})$ for a "<" node.

Lemma A.1 (side weights; Anderson et al., 2002, Lemma 2). *In every optimal tree, $sw(P) \geq sw(N)$ whenever P is the parent of N.*

Proof. There are four cases, each an order-preserving local modification that must not decrease the cost.

1. **P and N both "<".** Say N is the *no* child of P, with subtrees T_1 (the *yes* child of P), T_2 and T_3 (the children of N), of weights μ_1, μ_2, μ_3. A left rotation at P preserves key order, moves T_1 down one level and T_3 up one level, and leaves T_2 at the same depth. The cost changes by $\mu_1 - \mu_3 \geq 0$. Hence $sw(P) = \min(\mu_1, \mu_2 + \mu_3) \geq \min(\mu_2, \mu_3) = sw(N)$. The symmetric case is analogous.
2. **P "<" and N "$q = k$".** Moving the equality test above P lifts the leaf of k (μ_2) by one, sinks P's other subtree (μ_1) by one, and keeps the rest. The change is $\mu_1 - \mu_2 \geq 0$, so $sw(P) \geq \mu_2 = sw(N)$.
3. **Both equality tests.** Swapping them changes the cost by $sw(P) - sw(N) \geq 0$.
4. **P "$q = k$" and N "<".** Pushing the equality test below N, into the child that contains k, sinks the leaf of k by one and lifts the other child of N by one. This gives $\mu_1 \geq \min(\mu_2, \mu_3)$. ▫

These are the rotations used by Anderson et al.; unlike exchanges of non-adjacent nodes, they preserve the search-tree order.

Lemma A.2 (Chrobak et al., 2022, Lemma 1). *Some optimal tree is a leaf, or has a "<" root, or has an equality root testing a maximum-weight key of S.*

Proof. Let an optimal tree start with "$q = k_b$" and let k_a be a maximum-weight key. The parent P of the leaf of k_a can be changed into "$q = k_a$" without changing correctness or cost. If P is the root we are done. Otherwise Lemma A.1 along the root-to-P path gives $\beta_b = sw(\text{root}) \geq sw(P) = \beta_a$, so k_b is itself of maximum weight. ▫

Theorem 1 then follows from Lemma A.2 by the infinitesimal-perturbation argument of Chrobak et al. (2022, §3), which makes the chosen maximum-weight key unique. We do not re-derive it here.

APPENDIX B: NOTATION, PSEUDOCODE AND ADDITIONAL ILLUSTRATIONS

Table 6: Notation used throughout the paper.

Symbol	Meaning
$k_b,\ \beta_b$	key $b \in \{1..n\}$ and its weight (query frequency)
$g_a,\ \alpha_a$	gap (k_a, k_{a+1}), $a \in \{0..n\}$, and its weight; $k_0 = -\infty$, $k_{n+1} = +\infty$
$W,\ H$	total weight; Shannon entropy (bits) of the normalized element weights
$\pi,\ \text{rank}(b)$	keys sorted by non-decreasing weight (ties by index); $\text{rank}(b) = \pi^{-1}(b)$
$S(i,j,h)$	valid set (2): gaps $g_i..g_{j-1}$ and keys of $K(i,j)$ with rank $\leq h$
$K(i,j),\ m_{ij}$	key positions $\max(i,1)..j-1$ of interval (i,j); their number
$\text{opt}(i,j,h)$	minimum cost of a tree handling $S(i,j,h)$ (CGMY state)
$t,\ S_t(i,j)$	number of holes; the interval with its t heaviest keys removed (canonical state, §4.1)
$r_1 > \cdots > r_m$	ranks of the keys of $K(i,j)$ in decreasing order; $r_{m+1} = 0$
W_t	$w\big(S_t(i,j)\big)$, weight of the canonical state
$V(i,j,t)$	value of canonical state (i,j,t) (§4.1); $V = \text{opt}$ for $\tau = 0$
$t_L(b)$	holes of $S_t(i,j)$ lying left of k_b, i.e. in $K(i,b)$
τ	lower (admissibility) threshold for equality tests
λ^+	upper threshold: $3/7$ (SUCC), $1/2$ (FULL)
$\ell(i,j)$	length of the admissible removal chain of (i,j) (§4.3)
$C_c,\ L_{ij}$	dyadic weight class $\{b: \beta_b \in [2^c, 2^{c+1})\}$; number of classes meeting $K(i,j)$
$Q(\tau)$	$\lceil (2-\tau)/\tau \rceil - 1$, max. equality tests per class (Lemma 3)
R	ratio of the largest to the smallest positive key weight

```
Algorithm 1   HWP(β, α, variant, τ, λ⁺)                    ▷ returns OPT_τ and a tree
 1: ranks π; prefix sums of β, α; build multi-level segment tree     ▷ O(n log n)
 2: for all intervals (i,j): ℓ(i,j) ← CHAIN(i,j,τ)                ▷ pass 1, Lemma 2
 3:   allocate ℓ(i,j)+1 contiguous values per interval (row-major offsets)
 4: for d ← 1 … n+1 do                                            ▷ wavefront, pass 2
 5:          parallel for all intervals (i,j) with j − i = d do
 6:        (r_1..r_{ℓ+1}, W_0..W_ℓ) ← CHAIN(i,j,τ)            ▷ heaviest-first removals
 7:             need[t] ← not single(S_t) and W_t > 0 and not λ⁺-rule(t)
 8:       L_t ← +∞ for t = 0..ℓ          ▷ if ℓ < 4: state-by-state order (adaptive)
 9:      for b ← i+1 … j−1 do                                    ▷ interval-blocked order
10:                                                              tL ← 0
11:         for t ← 0 … ℓ do                          ▷ both child blocks read in order
12:                                                              if need[t]:
13:                    L_t ← min(L_t, V[i,b,min(tL,ℓ(i,b))] + V[b,j,min(t−tL,ℓ(b,j))])
14:                  if position of key removed at step t < b: tL ← tL + 1
15:      for t ← ℓ down to 0 do                       ▷ resolve equality chain, Eq. (5)
16:                        if single(S_t) or W_t = 0: V[i,j,t] ← 0
17:                else V[i,j,t] ← W_t + min(V[i,j,t+1] (t < ℓ), L_t (need[t]))
18: return V[root, 0]    (root = (0,n+1) in FULL, (1,n+1) in SUCC)
```

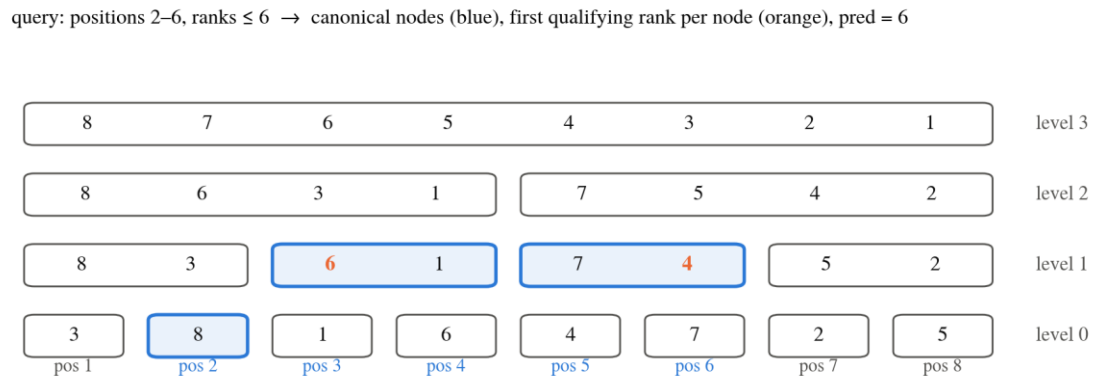

Figure 9: Multi-level segment (merge-sort) tree for eight keys. The query "positions 2–6, ranks ≤ 6" visits the three highlighted canonical nodes. A binary search in each node finds its first qualifying rank (orange), and the maximum of these is pred = 6.

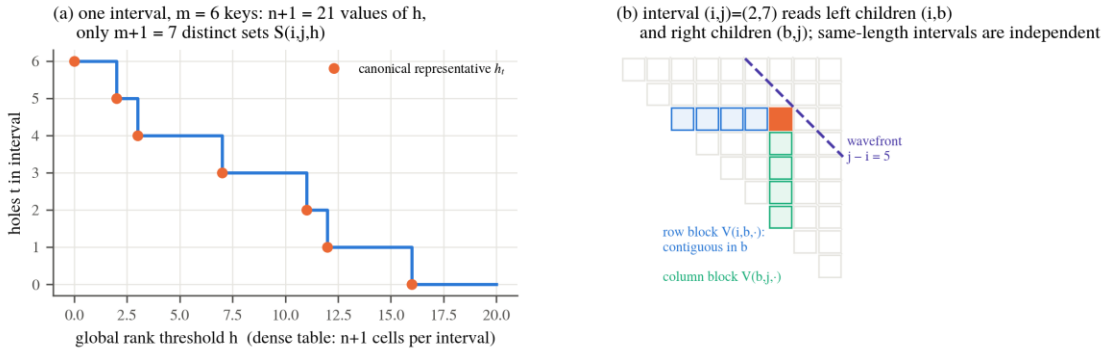

Figure 10: Canonical states and loop order. (a) For one interval with $m = 6$ keys, the $n + 1 = 21$ rank thresholds h of the dense CGMY table produce only $m + 1 = 7$ distinct sets. HWP stores one value per set, at its canonical representative. (b) Interval $(2,7)$ reads its left children $(2, b)$ (a contiguous row block) and right children $(b, 7)$. Intervals on the same anti-diagonal (equal length) are independent and form the parallel wavefront.

The complete pseudocode of every procedure, including the baselines, is in Supplementary S1.

APPENDIX C: GENERALITY AND ADDITIONAL RESULTS

Table 7: Which HWP components transfer to other dynamic programs.

Component	Requirement	Examples where it applies	Gain
Canonical states	states = interval minus globally ordered removals	2WCST (both variants), binary split trees, two-way classification	memory ≈ 1/3
Multi-level segment tree	weight/rank queries on range × threshold boxes	any weighted interval DP	$O(\log^2 n)$ state navigation
Interval-blocked loop + wavefront	interval DP with split minimization	Knuth-type OBST, matrix-chain, CYK-like, 2WCST	cache locality; O(n)-way parallelism
Weight-class thresholding	a threshold theorem (e.g. λ^-)	2WCST SUCC (exact); FULL (λ^+ only)	states $O(n^2 \log 2R)$
Additive dial (Thm. 6)	restricted family ⊇ entropy-bounded family	comparison trees with alphabetic constructions	anytime, ≤ OPT + 3W
Sub-cubic split minimum	QI / total monotonicity	three-way OBST (Knuth–Yao)	not available for 2WCST

Table 8: Time and peak memory, FULL variant (mean over the uniform and Zipf(1) families). The last column is the maximum relative gap of HWP-τ ($\tau = 0.25$) to the optimum where the optimum is known. "–" means not run: beyond the memory limit or impractical.

n	CGMY time	CGMY-vec time	HWP-E time	HWP-τ (τ=0.25) time	CGMY mem	HWP-E mem	HWP-τ mem	HWP-τ gap to OPT
200	347 ms	191 ms	340 ms	9 ms	36 MB	15 MB	5 MB	0.0000%
400	11.5 s	4.0 s	5.1 s	58 ms	251 MB	89 MB	6 MB	0.0002%
600	72.6 s	27.3 s	27.7 s	–	834 MB	286 MB	–	n/a
800	–	–	–	604 ms	–	–	11 MB	n/a
1200	–	–	448 s	–	–	2226 MB	–	n/a
1600	–	–	–	5.4 s	–	–	31 MB	n/a
3200	–	–	–	74.6 s	–	–	106 MB	n/a

Table 9: Three-way model: optimum (Knuth), the loss from power-of-two bucketing, and order-free lower bounds (mean over 3 seeds, $n = 400$).

Weights	n	OPT_3 / W (Knuth)	power-of-2 bucketing: excess	order-free Huffman / W (binary, LB for 2-way)	H / $\log_2 3$ (LB for 3-way)
clustered	400	7.065	0.55% (max 0.94%)	8.282	5.206
pareto	400	6.221	0.44% (max 0.74%)	8.123	5.106
uniform	400	7.516	0.87% (max 1.07%)	9.129	5.742
zipf1	400	5.804	0.34% (max 0.54%)	7.734	4.864
zipf2	400	3.371	0.17% (max 0.22%)	4.273	2.689

Table 10: Complete ablation ($n = 400$, Zipf(1), 5 seeds, both variants).

Variant	Configuration	time (mean ± sd)	peak memory	split evaluations	stored states	cost = OPT?
FULL	CGMY	11.711 ± 0.340 s	251 MB	2.18e+09	3.23e+07	✓
FULL	CGMY-vec	3.895 ± 0.217 s	251 MB	4.31e+09	3.23e+07	✓
FULL	HWP-E [adaptive, 2 threads]	2.785 ± 0.641 s	89 MB	2.18e+09	1.09e+07	✓
FULL	HWP-E [adaptive, λ^+]	4.733 ± 0.108 s	89 MB	2.18e+09	1.09e+07	✓
FULL	HWP-E [adaptive]	4.798 ± 0.072 s	89 MB	2.18e+09	1.09e+07	✓
FULL	HWP-E [blocked, mirror]	4.861 ± 0.191 s	173 MB	2.18e+09	1.09e+07	✓
FULL	HWP-E [blocked]	4.809 ± 0.245 s	89 MB	2.18e+09	1.09e+07	✓
FULL	HWP-E [t-outer, mirror]	20.359 ± 0.621 s	173 MB	2.18e+09	1.09e+07	✓
FULL	HWP-E [t-outer]	20.034 ± 0.845 s	89 MB	2.18e+09	1.09e+07	✓
FULL	HWP-τ (τ=0.1) [adaptive, chain: sort]	0.810 ± 0.050 s	7 MB	2.27e+07	2.01e+05	✓

Variant	Configuration	time (mean ± sd)	peak memory	split evaluations	stored states	cost = OPT?
FULL	HWP-τ (τ=0.1) [adaptive, λ^+]	0.126 ± 0.009 s	7 MB	2.26e+07	2.01e+05	✓
FULL	HWP-τ (τ=0.1) [adaptive]	0.126 ± 0.016 s	7 MB	2.27e+07	2.01e+05	✓
FULL	HWP-τ (τ=0.1) [blocked]	0.155 ± 0.019 s	7 MB	2.27e+07	2.01e+05	✓
FULL	HWP-τ (τ=0.1) [t-outer]	0.132 ± 0.023 s	7 MB	2.27e+07	2.01e+05	✓
FULL	HWP-τ (τ=0.25) [adaptive, chain: sort]	0.769 ± 0.052 s	6 MB	1.22e+07	1.05e+05	≈
FULL	HWP-τ (τ=0.25) [adaptive, λ^+]	0.061 ± 0.005 s	6 MB	1.21e+07	1.05e+05	≈
FULL	HWP-τ (τ=0.25) [adaptive]	0.068 ± 0.014 s	6 MB	1.22e+07	1.05e+05	≈
FULL	HWP-τ (τ=0.25) [blocked]	0.088 ± 0.004 s	6 MB	1.22e+07	1.05e+05	≈
FULL	HWP-τ (τ=0.25) [t-outer]	0.061 ± 0.005 s	6 MB	1.22e+07	1.05e+05	≈
SUCC	CGMY	11.228 ± 0.475 s	251 MB	2.13e+09	3.23e+07	✓
SUCC	CGMY-vec	3.763 ± 0.212 s	251 MB	4.28e+09	3.23e+07	✓
SUCC	HWP-E [adaptive, 2 threads]	3.006 ± 0.056 s	88 MB	2.13e+09	1.08e+07	✓
SUCC	HWP-E [adaptive, λ^+]	5.921 ± 0.272 s	88 MB	2.12e+09	1.08e+07	✓
SUCC	HWP-E [adaptive]	5.870 ± 0.184 s	88 MB	2.13e+09	1.08e+07	✓
SUCC	HWP-E [blocked, mirror]	5.703 ± 0.137 s	171 MB	2.13e+09	1.08e+07	✓
SUCC	HWP-E [blocked]	5.737 ± 0.259 s	88 MB	2.13e+09	1.08e+07	✓
SUCC	HWP-E [t-outer, mirror]	19.959 ± 0.451 s	171 MB	2.13e+09	1.08e+07	✓
SUCC	HWP-E [t-outer]	19.951 ± 0.807 s	88 MB	2.13e+09	1.08e+07	✓
SUCC	HWP-T [adaptive, chain: sort]	0.756 ± 0.029 s	6 MB	1.36e+07	1.29e+05	✓
SUCC	HWP-T [adaptive, λ^+]	0.072 ± 0.012 s	7 MB	1.31e+07	1.29e+05	✓
SUCC	HWP-T [adaptive]	0.068 ± 0.003 s	7 MB	1.36e+07	1.29e+05	✓
SUCC	HWP-T [blocked]	0.098 ± 0.005 s	7 MB	1.36e+07	1.29e+05	✓
SUCC	HWP-T [t-outer]	0.069 ± 0.003 s	7 MB	1.36e+07	1.29e+05	✓

Variant	Configuration	time (mean ± sd)	peak memory	split evaluations	stored states	cost = OPT?
SUCC	HWP-τ (τ=0.1) [adaptive, chain: sort]	0.850 ± 0.040 s	8 MB	3.15e+07	3.46e+05	✓
SUCC	HWP-τ (τ=0.1) [adaptive, λ^{+}]	0.189 ± 0.023 s	8 MB	3.09e+07	3.46e+05	✓
SUCC	HWP-τ (τ=0.1) [adaptive]	0.195 ± 0.021 s	8 MB	3.15e+07	3.46e+05	✓
SUCC	HWP-τ (τ=0.1) [blocked]	0.214 ± 0.025 s	8 MB	3.15e+07	3.46e+05	✓
SUCC	HWP-τ (τ=0.1) [t-outer]	0.194 ± 0.034 s	8 MB	3.15e+07	3.46e+05	✓