A Pedagogical Guide to Algorithm 1

CRPS-Optimal K-Partition via Dynamic Programming

Companion to *CRPS-Optimal Conformal Regression*

For an audience of CS/ML undergraduates

Contents

1 The Big Picture

1.1 Setting

You are given n labelled training examples $(x_1, y_1), \ldots, (x_n, y_n)$, sorted by the single feature x: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$. Each y_i is a real-valued response (e.g. a price, a temperature, a duration).

The goal is to cut the sorted sequence into K contiguous **bins**—groups of nearby points—so that within each bin the responses y are as homogeneous as possible. The bin boundaries are positions $0 = b_0 < b_1 < \cdots < b_K = n$: bin k contains points $b_{k-1} + 1$ through b_k.

Why bins?

Once the data is partitioned into bins, a new test point x^* falls into the bin whose x-range contains it. We then use the *empirical distribution* of the training y-values in that bin as a probabilistic prediction for y^*. Tight, homogeneous bins mean our prediction will be sharp and well-calibrated.

1.2 The Objective

We want to *measure* how good a candidate partition is, so we can search for the best one. The paper uses **leave-one-out (LOO) cross-validation** with CRPS (Continuous Ranked Probability Score) as the scoring rule.

For each bin, we hold out each point one at a time, build an empirical distribution from the remaining points in the bin, and measure how well it predicts the held-out value. Summing this across every point gives the *LOO-CRPS cost* of the partition.

We want to find the partition that minimises this total cost.

1.3 Why This Is Hard

The number of ways to place $K - 1$ cut points among $n - 1$ possible gap positions is $\binom{n-1}{K-1}$, which is exponential in K for large n. Evaluating all candidates is computationally infeasible.

Algorithm 1 solves this in $O(n^2 K)$ time—polynomial in both n and K— using **dynamic programming (DP)**.

2 The Cost of a Single Bin

2.1 Setup: Pairwise Absolute Sum

For a bin spanning sorted indices i through j (size $m = j - i + 1$), define the *pairwise absolute sum*:

$$W(i,j) = \sum_{\substack{\ell<r \\ \ell,r\in\{i,\ldots,j\}}} |y_\ell - y_r|.$$

This is a measure of the total *spread* of the y-values in the bin. All pairwise absolute differences are summed, so a bin with tightly clustered y-values gives a small W, and a bin with widely spread y-values gives a large W.

2.2 The LOO-CRPS Cost Formula

The total LOO-CRPS of the bin, summed over all m leave-one-out predictions, is:

$$\boxed{c(i,j) = \frac{m \cdot W(i,j)}{(m-1)^2}, \qquad m = j - i + 1.}$$

Intuition for this formula

- **Numerator** $m \cdot W$: total spread, scaled by bin size. More points and more spread both increase cost.
- **Denominator** $(m-1)^2$: a bias-correction factor. Larger bins are penalised less per-pair because each LOO prediction has $m-1$ atoms to work with—more data, better prediction. Without this correction, the objective would always prefer many tiny bins.
- **Minimum bin size**: a singleton ($m = 1$) has no leave-one-out partner, so the formula is undefined; by convention $c(i,i) = +\infty$. This enforces $m \geq 2$ throughout.
- **Perfectly homogeneous bin**: all y-values equal $\Rightarrow W = 0 \Rightarrow c = 0$. This is the theoretical lower bound.

2.3 Why CRPS?

The CRPS is a *strictly proper scoring rule*: it is uniquely minimised when your predictive distribution matches the true data-generating distribution. This means optimising LOO-CRPS produces partitions that are genuinely good for probabilistic prediction—not just ones that "look" homogeneous by some arbitrary distance measure.

3 Why Dynamic Programming?

3.1 The Optimal Substructure Property

Dynamic programming is applicable whenever a problem has *optimal substructure*: the optimal solution to the whole problem contains optimal solutions to sub-problems.

Optimal substructure of the partition problem

Claim: In the globally optimal K-partition of $\{1, \ldots, n\}$, the first k bins (for any $k < K$) form an optimal k-partition of $\{1, \ldots, b_k\}$.
Why: if they did not, you could substitute a better k-partition of the prefix, keeping the remaining $K - k$ bins unchanged, and strictly reduce the total cost—contradicting global optimality. This "cut-and-paste" argument is the standard proof of optimal substructure.

3.2 Consequence: A Recursive Formula

Define the DP value:

$\mathrm{dp}[k][j]$ = minimum total LOO-CRPS cost to partition observations $1, \ldots, j$ into exactly k bins.

By optimal substructure:

$$\mathrm{dp}[k][j] = \min_{k-1 \le i < j} \Big\{ \underbrace{\mathrm{dp}[k-1][i]}_{\text{first } i \text{ obs. in } k-1 \text{ bins}} + \underbrace{c(i+1,\, j)}_{\text{last bin: obs. } i+1 \text{ to } j} \Big\}.$$

Reading this: to partition the first j observations into k bins, try every possible start index $i + 1$ of the last bin. For each candidate, the last bin costs $c(i + 1, j)$ (precomputed), and the first i observations are handled by the already-solved sub-problem $\mathrm{dp}[k - 1][i]$. Take the minimum.

Base case ($k = 1$): there is only one way to put the first j observations in a single bin:

$$\mathrm{dp}[1][j] = c(1, j), \quad j = 2, \ldots, n.$$

The answer to the original problem is $\mathrm{dp}[K][n]$.

3.3 Comparison with Brute Force

Approach	Time	Notes
Enumerate all partitions	$O\binom{n}{K}$	Infeasible for $n = 1000$, $K = 10$
Greedy (sequential)	$O(n^2)$	Fast, but can miss global optimum
Dynamic programming	$O(n^2 K)$	Exact global optimum

The DP reuses previously computed sub-problem solutions. Each entry $\mathrm{dp}[k][j]$ is computed once and stored; filling it requires scanning $O(n)$ candidates i, each in $O(1)$ using precomputed costs.

4 Phase 1: Precomputing All Bin Costs

4.1 The Problem

There are $O(n^2)$ pairs (i, j) with $1 \le i \le j \le n$. We need $c(i, j)$ for all of them before the DP can run.

Naive approach: for each pair (i, j), scan all $\binom{m}{2}$ pairs (ℓ, r) to compute $W(i, j)$. This is $O(m^2)$ per entry and $O(n^4)$ overall—far too slow.

4.2 The Sweep Trick

Fix the left endpoint i and scan $j = i, i+1, \ldots, n$ from left to right. When you extend the bin from j to $j+1$, you add y_{j+1} to a set that currently contains $\{y_i, \ldots, y_j\}$. The pairwise sum increases by exactly:

$$\Delta W = \sum_{\ell=i}^{j} |y_\ell - y_{j+1}|.$$

This is the sum of absolute deviations of all current bin members from the new point y_{j+1}. Split the current members into those $\leq y_{j+1}$ and those $> y_{j+1}$:

$$\sum_{\ell : y_\ell \leq y_{j+1}} (y_{j+1} - y_\ell) = y_{j+1} \cdot r - S_{\leq},$$
$$\sum_{\ell : y_\ell > y_{j+1}} (y_\ell - y_{j+1}) = S_{>} - y_{j+1} \cdot (m - r),$$

so

$$\Delta W = y_{j+1} \cdot r - S_{\leq} + S_{>} - y_{j+1} \cdot (m - r),$$

where:

- r = number of current values $\leq y_{j+1}$ (the *rank* of y_{j+1}),
- $S_{\leq}$ = sum of current values $\leq y_{j+1}$,
- $S_{>}$ = sum of current values $> y_{j+1} = \Sigma - S_{\leq}$, with Σ the running total of all inserted values.

What we need at each step

Each time we insert y_{j+1} into the growing set, we need:

1. r: how many current values are $\leq y_{j+1}$? (*prefix count*)
2. $S_{\leq}$: what is their total? (*prefix sum*)

These are prefix queries over a dynamically growing set of values, evaluated at the query point y_{j+1}.

4.3 The Fenwick Tree (Binary Indexed Tree)

A **Fenwick tree** (also called a Binary Indexed Tree, or BIT) is a data structure that maintains an array and supports two operations, each in $O(\log n)$:

1. **Point update**: add a value at position p.
2. **Prefix query**: return the sum (or count) of all values at positions $1, \ldots, p$.

Two-tree construction. The algorithm uses two Fenwick trees in parallel:

- T_{cnt}: stores a *count* of 1 at each inserted y-value. A prefix query at y_{j+1} returns r (the rank).
- T_{sum}: stores the *value* y_ℓ at each inserted position. A prefix query at y_{j+1} returns $S_{\leq}$.

The trees are indexed by the *rank* of each y-value (coordinate-compressed to integers $1, \ldots, n$), so that the prefix-query endpoint y_{j+1} maps to its rank.

Per-step cost. Each insertion + two prefix queries costs $O(\log n)$. Over the outer loop (all n choices of i) and inner loop ($n - i$ steps), the total is $O(n^2 \log n)$.

Data structure	Query	Insert	Total precomputation
Sorted array	$O(\log n)$ binary search	$O(n)$ shift	$O(n^3)$
Fenwick tree	$O(\log n)$	$O(\log n)$	$O(n^2 \log n)$ ✓
Order-statistics BST	$O(\log n)$	$O(\log n)$	$O(n^2 \log n)$, larger constant

The Fenwick tree is the natural choice: simple, cache-friendly, and optimal up to constants.

Pseudocode for Phase 1.

Phase 1: Precompute cost matrix

for $i = 1$ **to** n **do**
  Initialise T_{cnt}, T_{sum}; running total $\Sigma \leftarrow 0$; $W \leftarrow 0$
  for $j = i$ **to** n **do**
    Insert y_j into T_{cnt} and T_{sum}; $\Sigma \leftarrow \Sigma + y_j$
    $r \leftarrow T_{\text{cnt}}.\text{prefixQuery}(y_j)$; $S_{\leq} \leftarrow T_{\text{sum}}.\text{prefixQuery}(y_j)$; $S_{>} \leftarrow \Sigma - S_{\leq}$
    $W \leftarrow W + y_j \cdot r - S_{\leq} + S_{>} - y_j \cdot (j - i + 1 - r)$
    $m \leftarrow j - i + 1$
    $c[i][j] \leftarrow$ **if** $m \geq 2$: $m\,W/(m-1)^2$ **else**: $+\infty$
  end for
end for

Important: fresh trees per row

The Fenwick trees are re-initialised for *each* value of i. The running W accumulates *only* the pairwise contributions from points $i, i+1, \ldots, j$ in the current row. Starting fresh at each i is essential for correctness.

5 Phase 2: Filling the DP Table

5.1 Algorithm

Phase 2: Fill DP table

for $j = 2$ **to** n **do**
  $\text{dp}[1][j] \leftarrow c[1][j]$; $\text{split}[1][j] \leftarrow 0$
end for
for $k = 2$ **to** K **do**

```
    for j = k to n do
        dp[k][j] ← +∞
        for i = k − 1 to j − 1 do
            v ← dp[k − 1][i] + c[i + 1][j]
            if v < dp[k][j] then
                dp[k][j] ← v;   split[k][j] ← i
            end if
        end for
    end for
end for
```

5.2 Step-by-Step Explanation

Base case ($k = 1$). When $k = 1$ there is no choice: all points $1, \ldots, j$ go into one bin. The cost is $c[1][j]$, which was precomputed in Phase 1. We store $\text{split}[1][j] = 0$ to indicate "the only bin starts at position 1."

Inductive step ($k \geq 2$). We fill the table layer by layer (increasing k), left-to-right within each layer (increasing j). By the time we compute $\text{dp}[k][j]$, all entries $\text{dp}[k-1][\cdot]$ are already in the table.

For each j, we try every candidate split point i (where the last bin starts at $i + 1$). The constraint $i \geq k - 1$ ensures the prefix $\{1, \ldots, i\}$ has enough points for $k - 1$ bins of size ≥ 2. The constraint $i < j$ ensures the last bin has at least one point (though c will enforce ≥ 2).

The split **table.** For each (k, j) we store $\text{split}[k][j] = \arg\min_i(\ldots)$—the index i at which the optimal last-bin cut was made. This is a standard DP *backtracking pointer*; without it we would know the optimal cost but not how to reconstruct the optimal partition.

5.3 Worked Toy Example

Suppose $n = 6$, $K = 3$. Entries already in the table when we compute dp[3][6]:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
$k = 1$	—	$c[1][2]$	$c[1][3]$	$c[1][4]$	$c[1][5]$	$c[1][6]$
$k = 2$	—	—	dp[2][3]	dp[2][4]	dp[2][5]	dp[2][6]
$k = 3$	—	—	—	—	—	**?**

Computing dp[3][6]: try all valid i:

$$i = 4: \quad \text{dp}[2][4] + c[5][6] \qquad i = 5: \quad \text{dp}[2][5] + c[6][6] = \text{dp}[2][5] + \infty \text{ (singleton!)}$$

(We also try $i = 2$ and $i = 3$ if they satisfy the constraint.) The minimum over all valid i gives dp[3][6].

Why is the singleton bin cost infinite?

A bin of size 1 cannot run leave-one-out: there would be zero training points to build the predictive distribution. The $+\infty$ cost automatically prevents singleton bins from appearing in any optimal solution.

6 Phase 3: Recovering the Bin Boundaries

6.1 Algorithm

Phase 3: Backtrack boundaries

$b_K \leftarrow n$; $k \leftarrow K$; $j \leftarrow n$
while $k \geq 1$ **do**
  $b_{k-1} \leftarrow \text{split}[k][j]$; $j \leftarrow b_{k-1}$; $k \leftarrow k-1$
end while
return $(b_0, b_1, \ldots, b_K)$

6.2 Explanation

After Phase 2, the table split$[k][j]$ records, for each sub-problem, where the optimal last-bin cut was made. Phase 3 reads these pointers backwards, starting from the full problem split$[K][n]$ and working down to the base case.

Trace.

1. Start: $b_K = n$. Read $b_{K-1} = \text{split}[K][n]$. The last bin is $\{b_{K-1}+1, \ldots, n\}$.
2. Move to sub-problem split$[K-1][b_{K-1}]$. Read b_{K-2}. The second-to-last bin is $\{b_{K-2}+1, \ldots, b_{K-1}\}$.
3. Continue until $k = 0$. At that point b_0 must equal 0 (all points assigned).

This is $O(K)$—negligible compared to the earlier phases.

7 Complexity Analysis

Phase	Outer loops	Inner work	Total
1: Precompute $c[i][j]$	$O(n^2)$ pairs	$O(\log n)$ per pair	$O(n^2 \log n)$
2: Fill DP table	K layers $\times$ $O(n)$ entries	$O(n)$ scan per entry	$O(n^2 K)$
3: Backtrack	—	$O(K)$ pointer reads	$O(K)$
Total			$O(n^2 K + n^2 \log n)$

The DP phase dominates when $K > \log n$ (the typical case in practice). Space is $O(n^2)$ for both the cost matrix and the DP table.

Potential speedup. If the cost function $c(i, j)$ satisfies the *quadrangle inequality* $c(a, c) + c(b, d) \leq c(a, d) + c(b, c)$ for $a \leq b \leq c \leq d$, then by the Knuth–Yao theorem the optimal split point is non-decreasing in j for each fixed k. This enables a divide-and-conquer strategy that reduces the DP to $O(nK)$. Whether our LOO-CRPS cost satisfies the quadrangle inequality remains an open problem.

8 Why Greedy Fails

A natural alternative is **greedy binary segmentation**: at each step, find the single cut point that reduces the total cost the most, add it to the partition, and repeat until K bins are formed.

Greedy does not find the global optimum in general

A locally best first cut may foreclose a globally better configuration of subsequent cuts. Once the first cut is placed, you are committed to it—even if a different first cut would have enabled $K - 1$ much better downstream cuts.

Example. Suppose $n = 6$, $K = 3$, and the y-values are:

$$y = (1,\ 2,\ 10,\ 11,\ 20,\ 21).$$

The optimal 3-partition is clearly $\{1, 2\}|\{10, 11\}|\{20, 21\}$ (three perfectly tight bins, total $W = 0$ up to constants). A greedy algorithm evaluating possible first cuts might find that splitting at position 3 (giving $\{1, 2, 10\}|\{11, 20, 21\}$) has the same first-step reduction as splitting at position 2. If it makes the wrong tie-breaking choice, it is stuck with a suboptimal partition. The DP evaluates *all combinations* simultaneously and cannot be trapped this way.

9 The Bigger Picture

Algorithm 1 is a subroutine in a *conformal prediction pipeline* for regression. Once the optimal K^*-partition is found:

1. A new test point x^* is assigned to the bin B_k whose x-range contains it.
2. The empirical CDF of $\{y_i : i \in B_k\}$ is used as a probabilistic predictive distribution for y^*.
3. A *conformal p-value* is computed using the CRPS as a nonconformity score, and the prediction set Γ^ε (a connected interval) is returned at any desired coverage level ε.

The key design coherence: the DP *trains* by minimising LOO-CRPS, and the conformal step *evaluates* using CRPS. Both stages speak the same probabilistic language—proper scoring rules.

Summary of the three algorithmic phases

1. **Phase 1** ($O(n^2 \log n)$): Precompute the LOO-CRPS cost $c[i][j]$ of every candidate bin, using a sweep with Fenwick trees to update the pairwise sum in $O(\log n)$ per step.
2. **Phase 2** ($O(n^2 K)$): Fill the DP table $\mathrm{dp}[k][j]$ layer by layer. Each entry is the globally optimal cost to partition the first j observations into k bins.
3. **Phase 3** ($O(K)$): Backtrack through the stored split-point pointers to recover the K optimal bin boundaries.