# Algorithm 1: CRPS-Optimal $K$-Partition

## A Pedagogical Walkthrough

## Dynamic Programming for Optimal Binning

Companion to *CRPS-Optimal Conformal Regression*

For CS/ML Undergraduates

# Outline

# What Are We Solving?

**Input**: $n$ training points $(x_i, y_i)$, sorted by $x$.

**Task**: partition into $K$ contiguous *bins*:

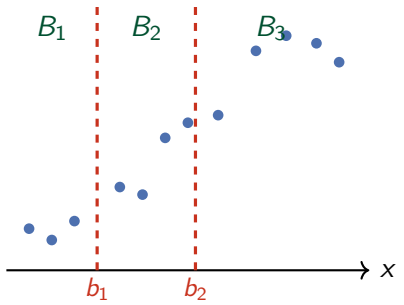
$$0 = b_0 < b_1 < \cdots < b_K = n$$

Bin $k$ = points $b_{k-1} + 1$ through $b_k$.

**For a new test point** $x^*$: find its bin, use the empirical CDF of the in-bin $y$-values as a probabilistic prediction.

**Goal**

Choose cut points so within-bin predictions are *as accurate as possible* under leave-one-out CRPS evaluation.




$K = 3$ bins; cut positions $b_1, b_2$ are what the algorithm finds.

# The Cost of a Bin

For bin spanning indices $i \dots j$ (size $m = j - i + 1$):

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

## Bin cost (Proposition 1)

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

= total leave-one-out CRPS of the bin

**Key properties**:

- $c = 0$ when all $y$-values are equal (ideal bin).
- $c(i,i) = +\infty$: singleton bins are forbidden.
- Denominator $(m-1)^2$: larger bins penalised less per pair.

## Why CRPS?

CRPS is a *strictly proper* scoring rule: uniquely minimised when your predictive distribution equals the true one. Optimising LOO-CRPS produces genuinely useful probabilistic predictions, not just visually homogeneous groups.

**Total objective to minimise**:

$$\sum_{k=1}^{K} c(b_{k-1}+1,\ b_k)$$

over all choices of $b_1, \dots, b_{K-1}$.

# Why Dynamic Programming?

**Brute force**: $\binom{n-1}{K-1}$ candidate partitions.
For $n = 1000$, $K = 10$: $\approx 10^{24}$. Infeasible.

**Key insight — optimal substructure**:

**Proposition (optimal substructure)**

In the globally optimal $K$-partition, the *first $k$ bins* form an optimal $k$-partition of $\{1, \ldots, b_k\}$ for every $k < K$.

**Proof sketch**: if the prefix were sub-optimal, replace it with a better one, keeping the suffix fixed. Total cost strictly decreases — contradicting global optimality.

This "cut-and-paste" argument is the standard proof of optimal substructure in all DP

**DP table**: $\mathrm{dp}[k][j]$ = min cost to partition $\{1, \ldots, j\}$ into $k$ bins.

**Recurrence**:

$$\mathrm{dp}[k][j] = \min_{k-1 \le i < j} \{\mathrm{dp}[k-1][i] + c(i+1, j)\}$$

Try every "last-bin" start point $i+1$: first $i$ obs. handled by already-solved sub-problem $\mathrm{dp}[k-1][i]$.

**Base case**: $\mathrm{dp}[1][j] = c(1, j)$.

**Answer**: $\mathrm{dp}[K][n]$; boundaries via backtracking.

# Algorithm Overview

## Three phases (Algorithm 1)

1. **Precompute** all $O(n^2)$ bin costs $c[i][j]$ — $O(n^2 \log n)$ using Fenwick trees
2. **Fill DP table** $\mathrm{dp}[k][j]$ for $k = 1, \ldots, K$ and $j = 1, \ldots, n$ — $O(n^2 K)$
3. **Backtrack** split-point pointers to recover boundaries — $O(K)$

**Phase 1 output**:
A cost matrix

$$c[i][j] \text{ for all } i \leq j$$

stored as an $n \times n$ table.

**Phase 2 output**:
Two $K \times n$ tables:

$$\mathrm{dp}[k][j], \quad \mathrm{split}[k][j]$$

where $\mathrm{split}$ stores the argmin.

**Phase 3 output**:
Boundaries

$$b_0{=}0 < b_1 < \cdots < b_K{=}n$$

recovered in $K$ pointer reads.

# Phase 1 — The Sweep Trick

## Phase 1 pseudocode

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

**Key idea — sweep from left:**
Fix $i$; grow the bin one point at a time. When $y_{j+1}$ joins the bin, $W$ increases by:

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

Split into values $\leq y_{j+1}$ and $> y_{j+1}$:

$$\Delta W = \underbrace{y_{j+1} r - S_{\leq}}_{\text{below}} + \underbrace{S_{>} - y_{j+1}(m-r)}_{\text{above}}$$

$r$ = rank of $y_{j+1}$, $S_{\leq}/S_{>}$ = sums below/above.

### Two queries per step

- $r$ = prefix *count* up to $y_{j+1}$
- $S_{\leq}$ = prefix *sum* up to $y_{j+1}$

# Phase 1 — The Fenwick Tree

**Fenwick tree (Binary Indexed Tree)**:

A compact array supporting two operations in $O(\log n)$:

1. **Point update**: add a value at position $p$.
2. **Prefix query**: return total at positions $1, \ldots, p$.

**Two trees in parallel**:

| Tree | Stores | Query returns |
|---|---|---|
| $T_{\text{cnt}}$ | count at rank($y$) | $r$ (rank) |
| $T_{\text{sum}}$ | value $y$ at rank($y$) | $S_{\leq}$ |

$S_{>} = \Sigma - S_{\leq}$ (from running total $\Sigma$).

**Reset per row**: trees re-initialised for each $i$.

**Why not simpler structures?**

| Structure | Insert | Query |
|---|---|---|
| Sorted array | $O(n)$ | $O(\log n)$ |
| Fenwick tree | $O(\log n)$ | $O(\log n)$ |
| BST | $O(\log n)$ | $O(\log n)$ |

**Complexity**

$O(n^2)$ pairs $\times$ $O(\log n)$ per pair
$= O(n^2 \log n)$ total

Fenwick trees are cache-friendly and require no pointers or rebalancing.

**Coordinate compression**

Trees are indexed by *rank* of $y$, not raw value, to

# Phase 2 — Filling the DP Table

## Phase 2 pseudocode

```
for j = 2 to n do                                   ▷ base case
    dp[1][j] ← c[1][j]
    split[1][j] ← 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
```

**Base case** ($k = 1$): one bin for all $1, \ldots, j$. Cost $= c[1][j]$. Split pointer $= 0$.

**Inductive step** ($k \geq 2$): For each $j$, scan split points $i$:

$$\underbrace{\mathrm{dp}[k-1][i]}_{\substack{\text{first } i \text{ pts} \\ \text{in } k-1 \text{ bins}}} + \underbrace{c[i+1][j]}_{\text{last bin}}$$

Store argmin in $\mathrm{split}[k][j]$ for backtracking.

### Fill order matters

Layer $k$ uses layer $k-1$ (already complete). Column $j$ uses $\mathrm{dp}[k-1][i]$ for $i < j$ (already computed). No circular dependencies.

# Phase 2 — Worked Example ($n$=6, $K$=3)

**Computing** $\mathrm{dp}[3][6]$:
Try all split points $i$:

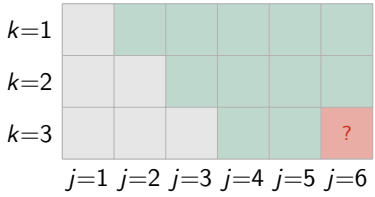
$$\mathrm{dp}[3][6] = \min_{i=2}^{5} \{\mathrm{dp}[2][i] + c[i+1][6]\}$$

| $i$ | $\mathrm{dp}[2][i] + c[i+1][6]$ | note |
|---|---|---|
| 2 | $\mathrm{dp}[2][2] + c[3][6]$ | |
| 3 | $\mathrm{dp}[2][3] + c[4][6]$ | |
| 4 | $\mathrm{dp}[2][4] + c[5][6]$ | |
| 5 | $\mathrm{dp}[2][5] + c[6][6]$ | $= +\infty$ (singleton!) |

The minimising $i$ is stored as $\mathrm{split}[3][6]$.

**Why is** $c[6][6] = +\infty$**?**
A bin of size 1 has no leave-one-out partner. The $+\infty$ automatically prevents singleton bins from appearing in any optimal solution — no explicit feasibility check needed.

**DP table structure**:



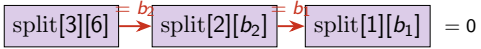

Green cells computed; grey = infeasible ($< k$ points for $k$ bins); red = $\mathrm{dp}[3][6]$ we are filling.

# Phase 3 — Recovering the Boundaries

**Phase 3 pseudocode**

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

**Pointer chain** for $K = 3$, $n = 6$:




**How it works**:

1. Start at $\mathrm{split}[K][n]$. This gives $b_{K-1}$: where the last bin starts.
2. Move to $\mathrm{split}[K-1][b_{K-1}]$. This gives $b_{K-2}$.
3. Continue until $b_0 = 0$ is reached.

Each pointer read is $O(1)$; total $O(K)$.

**Correctness**

Each pointer stores the *exact* argmin from Phase 2. Following the chain reconstructs the globally optimal partition without re-evaluating any costs.

**Bin layout recovered**:

$B_1 = \{1, \ldots, b_1\}$, $B_2 = \{b_1+1, \ldots, b_2\}$,

# Complexity Summary

| Phase | Time | Space |
|---|---|---|
| **1: Precompute** | $O(n^2 \log n)$ | $O(n^2)$ |
| **2: DP fill** | $O(n^2 K)$ | $O(n^2)$ |
| **3: Backtrack** | $O(K)$ | $O(1)$ |
| **Total** | $O(n^2 K + n^2 \log n)$ | $O(n^2)$ |

**Context**: for $n = 1000$, $K = 10$:

- Phase 1: $\approx 10^7$ operations.
- Phase 2: $\approx 10^7$ operations.
- Both tractable on a laptop in under a second.

## Potential speedup: Knuth–Yao theorem

If $c(i, j)$ 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 the optimal split point is *non-decreasing in* $j$. This enables divide-and-conquer filling of the DP table in $O(nK)$ instead of $O(n^2 K)$.

Whether the LOO-CRPS cost satisfies the QI is an **open problem**.

# Why Greedy Fails

**Greedy binary segmentation**: at each step, add the single cut that reduces total cost the most; repeat until $K$ bins are formed.

**Greedy is not globally optimal**

A locally best first cut may foreclose a much better configuration of subsequent cuts.

**Toy example**: $y = (1, 2, 10, 11, 20, 21)$, $K = 3$.
Optimal: $\{1, 2\} \mid \{10, 11\} \mid \{20, 21\}$ — three tight bins.

Greedy might cut $\{1, 2, 10\}$ vs. $\{11, 20, 21\}$, leaving a suboptimal remainder (wrong grouping of 10 and 11).

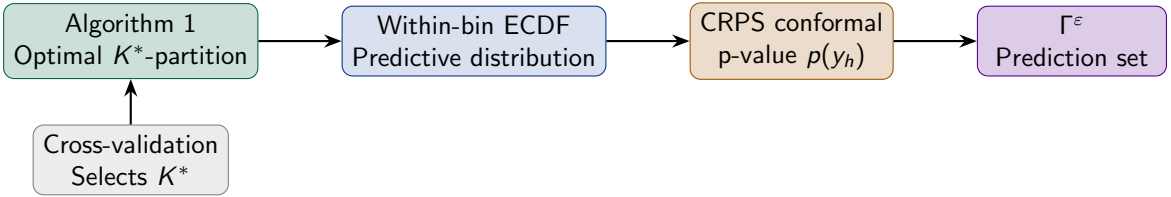
**What the DP does instead**:

It fills the table for *all* sub-problem sizes simultaneously. When it computes $\mathrm{dp}[3][6]$, it has already correctly solved $\mathrm{dp}[2][2]$, $\mathrm{dp}[2][3]$, $\mathrm{dp}[2][4]$, $\mathrm{dp}[2][5]$ — the best 2-bin prefix for every prefix length. It then picks the prefix that, combined with the cheapest last bin, gives the global minimum.

**Analogy: shortest paths**

Dijkstra's algorithm is correct because it expands nodes in optimal-cost order. Our DP is correct because it fills sub-problems in increasing size order, each solved *exactly* before it is used.

# Algorithm 1 in the Full Pipeline




**Design coherence**:
The DP *trains* by minimising LOO-CRPS.
The conformal step *evaluates* using CRPS.
Both stages speak the same probabilistic language.

**Validity guarantee**:
Under exchangeability within each bin:
$\mathbb{P}(y^* \in \Gamma^\varepsilon) \geq 1 - \varepsilon$

**Key properties of $\Gamma^\varepsilon$**:

- Always a *connected interval*: the CRPS nonconformity score is convex in $y_h$, so its sublevel sets are intervals.
- Adapts to heteroscedasticity: wider bins in high-variance regions, narrower in low-variance regions.

# Summary

## Algorithm 1 in three lines

1. **Precompute** $c[i][j]$ for all bins in $O(n^2 \log n)$ via a left-to-right sweep with two Fenwick trees.
2. **Fill** the DP table $\mathrm{dp}[k][j]$ layer by layer in $O(n^2 K)$: each entry is the globally optimal cost to partition $\{1, \ldots, j\}$ into $k$ bins.
3. **Backtrack** the split-point pointers in $O(K)$ to recover the exact bin boundaries.

**What makes this work**:

- Closed-form LOO-CRPS cost (Proposition 1)
- Optimal substructure (Proposition 3)
- Fenwick tree for $O(\log n)$ incremental updates

**Open questions**:

- Does $c(i, j)$ satisfy the quadrangle inequality? ($O(nK)$ speedup available if yes)
- Consistency of cross-validated $K^*$ as $n \to \infty$?
- Extension to $d > 1$ covariates?