Addition Strategies: Chunking by Bases and Ones

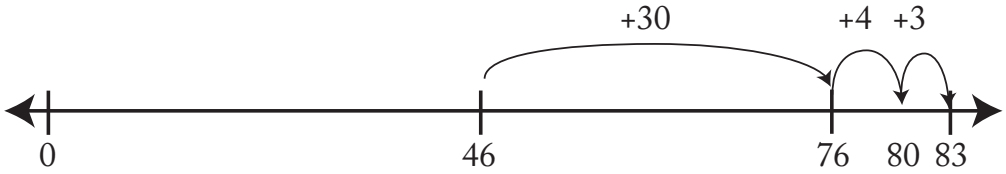
Compiled by: Theodore M. Savich

March 28, 2025

Transcript

Strategy descriptions and examples adapted from Hackenberg (2025). Problem: Max has 46 comic books. For his birthday, his father gives him 37 more comic books. How many comic books does Max have now?

Dionne's solution: "He has 46. Then 37 more. [She writes down 46, 76.] That's the 30. And then 7 more. Well, 4 more makes 80, and then I only need to do 3 more, 83."



Notation Representing Sarah's Solution:

$$\begin{aligned} 46 + 37 &= \square \\ 46 + 30 &= 76 \\ 76 + 4 &= 80 \\ 80 + 3 &= 83 \end{aligned}$$

Description of Strategy:

Objective: Begin with one number. Then, break the other number down into bases and units. In COBO, you count on each base individually - then the ones. With Chunking, instead of adding each base individually, add them in well-chosen, larger groups. Likewise, combine the units in groups rather than one by one—though there are instances when adding a single base or unit makes strategic sense. The overall goal is to create larger, intentional groupings, and it's important to clarify why each grouping is considered strategic. Usually, the goal with chunking on ones is to make a base first, then you can chunk on the rest of the ones. Usually when chunking on the bases, the goal is to make a base-of-bases first (so, in base ten, the goal would be to try and make one hundred), because then you can chunk on the rest of the bases (and ones) all at once.

Description of Strategy

- **Objective:** Similar to COBO but add bases and ones in larger, strategic chunks.
- **Example:** $46 + 37$
  - Start at 46.

- Add all tens at once: $46 + 30 = 76$.
- Add ones strategically: $76 + 4 = 80$, then $80 + 3 = 83$.

Automaton Type

Finite State Automaton (FSA) with basic arithmetic capability.

Formal Description of the Automaton

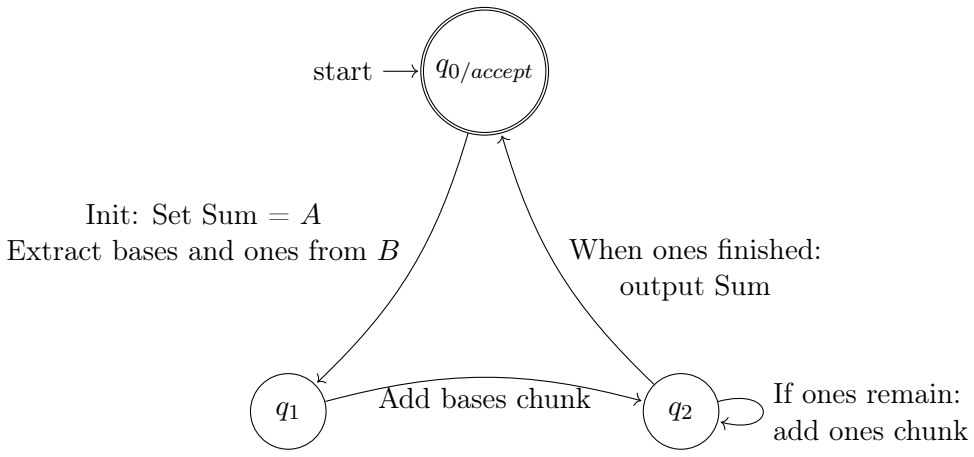
We define the automaton as the tuple

$$M = (Q, \Sigma, \delta, q_{0/accept}, F)$$

where:

- $Q = \{q_{0/accept}, q_1, q_2\}$ is the set of states.
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +\}$ is the input alphabet.
- $q_{0/accept}$ is the start state, which is also the accept state.
- $F = \{q_{0/accept}\}$ is the set of accepting states.
- The transition function δ is defined as:
  1. $\delta(q_{0/accept}, \text{“}A, B\text{”}) = q_1$ with the action: set Sum $\leftarrow A$ and extract the base and ones chunks from B.
  2. $\delta(q_1, \varepsilon) = q_2$ with the action: update Sum $\leftarrow$ Sum$+$ (the bases chunk from B).
  3. $\delta(q_2, \varepsilon) = q_2$ with the action: if ones remain, add a strategic ones chunk to Sum (loop as needed).
  4. $\delta(q_2, \varepsilon) = q_{0/accept}$ with the action: when ones are finished, output Sum.

Automaton Diagram for Chunking by Bases and Ones



HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Addition Strategies: Chunking by Bases and Ones</title>
    <style>
        body { font-family: sans-serif; }
        #diagramChunkingSVG { border: 1px solid #d3d3d3; }
        #outputContainer { margin-top: 20px; }
        .number-line-tick { stroke: black; stroke-width: 1; }
        .number-line-break { stroke: black; stroke-width: 1; stroke-dasharray: 5 5;} /*
            For scale break */
        .number-line-label { font-size: 12px; text-anchor: middle; } /* Centered labels */
        .jump-arrow { fill: none; stroke: green; stroke-width: 2; } /* Changed color */
        .jump-arrow-head { fill: green; stroke: green; } /* Changed color */
        .jump-label { font-size: 12px; text-anchor: middle; fill: green; } /* Changed
            color */
        .stopping-point { fill: red; stroke: black; stroke-width: 1; }
         /* Number line arrowhead */
          .number-line-arrow { fill: black; stroke: black;}
    </style>
</head>
<body>

<h1>Addition Strategies: Chunking by Bases and Then Ones</h1>

<div>
    <label for="chunkingAddend1">Addend 1:</label>
    <input type="number" id="chunkingAddend1" value="46">
</div>
<div>
    <label for="chunkingAddend2">Addend 2:</label>
    <input type="number" id="chunkingAddend2" value="37">
</div>

<button onclick="runChunkingAutomaton()">Calculate and Visualize</button>

<div id="outputContainer">
    <h2>Explanation:</h2>
    <div id="chunkingOutput">
        <!-- Text output will be displayed here -->
    </div>
</div>

<h2>Diagram:</h2>
<svg id="diagramChunkingSVG" width="700" height="350"></svg>

<script>
    document.addEventListener('DOMContentLoaded', function() {
        const outputElement = document.getElementById('chunkingOutput');
        const chunkingAddend1Input = document.getElementById('chunkingAddend1');
        const chunkingAddend2Input = document.getElementById('chunkingAddend2');
        const diagramChunkingSVG = document.getElementById('diagramChunkingSVG');
```

```
51
52          if (!outputElement || !diagramChunkingSVG) {
53              console.warn('Element chunkingOutput or diagramChunkingSVG not found');
54              return;
55          }
56
57          window.runChunkingAutomaton = function() {
58              try {
59                  const addend1 = parseInt(chunkingAddend1Input.value);
60                  const addend2 = parseInt(chunkingAddend2Input.value);
61                  if (isNaN(addend1) || isNaN(addend2)) {
62                      outputElement.textContent = 'Please enter valid numbers for both
                            addends';
63                      return;
64                  }
65
66                  let output = `<h2>Chunking by Bases and Ones (Flexible)</h2>\n\n`;
67                  output += `<p><strong>Problem:</strong> ${addend1} + ${addend2}</p>\n\n`;
68
69                  let tensToAddTotal = Math.floor(addend2 / 10) * 10;
70                  let onesToAddTotal = addend2 % 10;
71
72                  output += `Step 1: Split ${addend2} into ${tensToAddTotal} (tens) + ${
                        onesToAddTotal} (ones)\n\n`;
73
74                  let currentSum = addend1;
75                  const chunkSteps = [];
76                  let stepCounter = 2;
77
78                  // --- Strategy Decision: Add Ones First or Tens First? ---
79                  const addOnesFirstDecision = Math.random() < 0.3; // 30% chance to add ones
                         first (if possible)
80                  let onesAddedFirst = false;
81
82                  if (addOnesFirstDecision && onesToAddTotal > 0) {
83                      // Try adding ones first to make the next ten
84                      const onesToNextTenInitial = (10 - (currentSum % 10)) % 10;
85                      if (onesToNextTenInitial > 0 && onesToAddTotal >= onesToNextTenInitial)
                             {
86                          output += `Step ${stepCounter}: Add ones chunk first to make a ten
                                \n`;
87                          stepCounter++;
88                          chunkSteps.push({
89                             from: currentSum,
90                             to: currentSum + onesToNextTenInitial,
91                             label: `+${onesToNextTenInitial}`
92                         });
93                          output += `<p>${currentSum} + ${onesToNextTenInitial} = ${
                                currentSum + onesToNextTenInitial} (Making the next ten)</p>\n`;
94                          currentSum += onesToNextTenInitial;
95                          onesToAddTotal -= onesToNextTenInitial;
96                          onesAddedFirst = true; // Flag that we adjusted ones already
97                          output += '\n';
98                      }
```

```
        }

        // --- Tens Chunking (Potentially after adding some ones) ---
        if (tensToAddTotal > 0) {
            output += `Step ${stepCounter}: Add tens chunk(s)\n`;
            stepCounter++;

            while (tensToAddTotal > 0) {
                // Calculate tens needed to reach the *next* hundred
                let amountToNextHundred = (currentSum % 100 === 0) ? 0 : 100 - (currentSum % 100);
                let tensToNextHundred = Math.floor(amountToNextHundred / 10) * 10;

                let tensChunk = 0;

                if (tensToNextHundred > 0 && tensToAddTotal >= tensToNextHundred) {
                    // Option 1: Chunk exactly to the next hundred
                    tensChunk = tensToNextHundred;
                    output += `<p>${currentSum} + ${tensChunk} = ${currentSum + tensChunk} (Making the next hundred)</p>\n`;
                } else {
                    // Option 2: Add remaining tens, or a smaller "honest" chunk if large amount remains
                    if (tensToAddTotal <= 30 || Math.random() < 0.6) { // More likely to add all if 30 or less, or 60% chance otherwise
                        tensChunk = tensToAddTotal; // Add all remaining tens
                        output += `<p>${currentSum} + ${tensChunk} = ${currentSum + tensChunk}</p>\n`;
                    } else {
                        // Add a smaller "honest" chunk (e.g., 10, 20, or 30) - more random choices possible here
                        tensChunk = (Math.floor(Math.random() * 3) + 1) * 10; // Randomly 10, 20, or 30
                        tensChunk = Math.min(tensChunk, tensToAddTotal); // Don't add more than available
                        output += `<p>${currentSum} + ${tensChunk} = ${currentSum + tensChunk}</p>\n`;
                    }
                }

                if (tensChunk > 0) {
                    chunkSteps.push({
                        from: currentSum,
                        to: currentSum + tensChunk,
                        label: `+${tensChunk}`
                    });
                    currentSum += tensChunk;
                    tensToAddTotal -= tensChunk;
                } else {
                    // Safety break if something went wrong
                    break;
                }
            }
            output += '\n';
```

```
144                 }
145
146                 // --- Remaining Ones Chunking (If not added first or some left over) ---
147                 if (onesToAddTotal > 0) {
148                     output += `Step ${stepCounter}: Add remaining ones chunk(s)\n`;
149
150                     // Strategic ones (make next ten) - might happen again if tens landed
                        awkwardly
151                     const onesToNextTen = (10 - (currentSum % 10)) % 10;
152
153                     if (onesToNextTen > 0 && onesToAddTotal >= onesToNextTen) {
154                         // Chunk 1: Reach the next ten
155                         chunkSteps.push({
156                             from: currentSum,
157                             to: currentSum + onesToNextTen,
158                             label: `+${onesToNextTen}`
159                         });
160                         output += `<p>${currentSum} + ${onesToNextTen} = ${currentSum +
                            onesToNextTen} (Making the next ten)</p>\n`;
161                         currentSum += onesToNextTen;
162                         onesToAddTotal -= onesToNextTen;
163
164                         // Chunk 2: Add the rest
165                         if (onesToAddTotal > 0) {
166                             chunkSteps.push({
167                                 from: currentSum,
168                                 to: currentSum + onesToAddTotal,
169                                 label: `+${onesToAddTotal}`
170                             });
171                             output += `<p>${currentSum} + ${onesToAddTotal} = ${currentSum +
                                onesToAddTotal}</p>\n`;
172                             currentSum += onesToAddTotal;
173                             onesToAddTotal = 0;
174                         }
175                     } else if (onesToAddTotal > 0) {
176                         // Add all remaining ones
177                         chunkSteps.push({
178                             from: currentSum,
179                             to: currentSum + onesToAddTotal,
180                             label: `+${onesToAddTotal}`
181                         });
182                         output += `<p>${currentSum} + ${onesToAddTotal} = ${currentSum +
                            onesToAddTotal}</p>\n`;
183                         currentSum += onesToAddTotal;
184                         onesToAddTotal = 0;
185                     }
186                     output += '\n';
187                 }
188
189
190                 output += `Result: ${addend1} + ${addend2} = ${currentSum}`;
191                 outputElement.innerHTML = output;
192                 typesetMath();
193
```

```
            drawChunkingNumberLineDiagram('diagramChunkingSVG', addend1, addend2,
                chunkSteps, currentSum);

        } catch (error) {
            outputElement.textContent = `Error: ${error.message}`;
        }
    };

    // drawChunkingNumberLineDiagram function remains the same
    // ... (Keep the FULL drawChunkingNumberLineDiagram function and its helpers from
        previous responses) ...
     function drawChunkingNumberLineDiagram(svgId, addend1, addend2, chunkSteps,
         finalSum) {
        const svg = document.getElementById(svgId);
        if (!svg) return;
        svg.innerHTML = '';

        const svgWidth = parseFloat(svg.getAttribute('width'));
        const svgHeight = parseFloat(svg.getAttribute('height'));
        const startX = 50;
        const endX = svgWidth - 50;
        const numberLineY = svgHeight / 2 + 30; // Lower number line slightly
        const tickHeight = 10;
        const labelOffsetBase = 20;
        const jumpHeightLarge = 60; // Increased height for larger jumps
        const jumpHeightSmall = 40; // Height for smaller jumps (ones chunks)
        const jumpLabelOffset = 15;
        const arrowSize = 5;
        const scaleBreakThreshold = 40; // Adjust if needed

        // Draw Number Line & 0 Tick
        const numberLine = document.createElementNS('http://www.w3.org/2000/svg', '
            line');
        numberLine.setAttribute('x1', startX);
        numberLine.setAttribute('y1', numberLineY);
        numberLine.setAttribute('x2', endX);
        numberLine.setAttribute('y2', numberLineY);
        numberLine.setAttribute('class', 'number-line-tick');
        svg.appendChild(numberLine);

        const zeroTick = document.createElementNS('http://www.w3.org/2000/svg', 'line'
            );
        zeroTick.setAttribute('x1', startX);
        zeroTick.setAttribute('y1', numberLineY - tickHeight / 2);
        zeroTick.setAttribute('x2', startX);
        zeroTick.setAttribute('y2', numberLineY + tickHeight / 2);
        zeroTick.setAttribute('class', 'number-line-tick');
        svg.appendChild(zeroTick);
        createText(svg, startX, numberLineY + labelOffsetBase, '0', 'number-line-label
            ');

        // Calculate scale and handle potential break
        let displayRangeStart = 0;
        let scaleStartX = startX;
```

```
242             let drawScaleBreak = false;
243
244             // Determine the actual min and max values shown *after* the break
245             let minValAfterBreak = addend1;
246             let maxValAfterBreak = finalSum;
247             chunkSteps.forEach(step => {
248                 minValAfterBreak = Math.min(minValAfterBreak, step.from, step.to);
249                 maxValAfterBreak = Math.max(maxValAfterBreak, step.from, step.to);
250             });
251
252
253             if (addend1 > scaleBreakThreshold) {
254                 displayRangeStart = minValAfterBreak - 10; // Start range slightly before
                        min value shown after break
255                 scaleStartX = startX + 30; // Leave space for break symbol
256                 drawScaleBreak = true;
257                 drawScaleBreakSymbol(svg, scaleStartX - 15, numberLineY); // Draw break
                        symbol
258             } else {
259                 displayRangeStart = 0; // Start from 0 if no break
260             }
261
262             const displayRangeEnd = maxValAfterBreak + 10; // End range slightly after max
                     value shown
263             const displayRange = Math.max(displayRangeEnd - displayRangeStart, 1); //
                    Avoid division by zero if range is 0
264             const scale = (endX - scaleStartX) / displayRange;
265
266             // Function to convert value to X coordinate based on scale
267             function valueToX(value) {
268                 if (value < displayRangeStart && drawScaleBreak) {
269                     // Values before the effective start are compressed near the break
                            symbol
270                     return scaleStartX - 10; // Place them just before the break starts
                            visually
271                 }
272                  // Ensure values stay within the visible range after the break starts
273                 const scaledValue = scaleStartX + (value - displayRangeStart) * scale;
274                 return Math.min(scaledValue, endX); // Cap at endX
275             }
276
277             // Draw Ticks and Labels for relevant points
278             function drawTickAndLabel(value, index) {
279                 const x = valueToX(value);
280                 if (x < scaleStartX - 5 && value !== 0) return; // Don't draw ticks in
                        compressed area unless it's 0 or very close to break
281
282                 const tick = document.createElementNS('http://www.w3.org/2000/svg', 'line')
                        ;
283                 tick.setAttribute('x1', x);
284                 tick.setAttribute('y1', numberLineY - tickHeight / 2);
285                 tick.setAttribute('x2', x);
286                 tick.setAttribute('y2', numberLineY + tickHeight / 2);
287                 tick.setAttribute('class', 'number-line-tick');
```

```
        svg.appendChild(tick);
        const labelOffset = labelOffsetBase * (index % 2 === 0 ? 1 : -1.5);
        createText(svg, x, numberLineY + labelOffset, value.toString(), 'number-line-label');
    }

    drawTickAndLabel(addend1, 0); // Starting addend
    let lastToValue = addend1;

    // Draw chunk jumps
    chunkSteps.forEach((step, index) => {
        const x1 = valueToX(step.from);
        const x2 = valueToX(step.to);
         // Check if both start and end points are significantly beyond the SVG width
         if(x1 >= endX - 1 && x2 >= endX - 1) return;

        // Determine jump height based on chunk size (e.g., tens vs ones)
        const isLargeChunk = Math.abs(step.to - step.from) >= 10; // Define what constitutes a "large" chunk
        const currentJumpHeight = isLargeChunk ? jumpHeightLarge : jumpHeightSmall;
        const staggerOffset = index % 2 === 0 ? 0 : currentJumpHeight * 0.5; // Stagger jump height slightly

        createJumpArrow(svg, x1, numberLineY, x2, numberLineY, currentJumpHeight + staggerOffset);
        createText(svg, (x1 + x2) / 2, numberLineY - (currentJumpHeight + staggerOffset) - jumpLabelOffset, step.label, 'jump-label');
        drawTickAndLabel(step.to, index + 1);
        lastToValue = step.to;
    });

    // Ensure final sum tick is drawn if it wasn't the last 'to' value and is within range
    if (finalSum !== lastToValue && valueToX(finalSum) <= endX) {
        drawTickAndLabel(finalSum, chunkSteps.length + 1);
    }

     // Add arrowhead to the right end of the visible number line segment
    const endLineX = valueToX(displayRangeEnd); // Use the calculated end based on scaling
    const mainArrowHead = document.createElementNS('http://www.w3.org/2000/svg', 'path');
    mainArrowHead.setAttribute('d', `M ${endLineX - arrowSize} ${numberLineY - arrowSize/2} L ${endLineX} ${numberLineY} L ${endLineX - arrowSize} ${numberLineY + arrowSize/2} Z`);
    mainArrowHead.setAttribute('class', 'number-line-arrow');
    svg.appendChild(mainArrowHead);

    // Start point marker
    drawStoppingPoint(svg, valueToX(addend1), numberLineY, 'Start');


    // --- Helper SVG drawing functions --- (Keep these the same) ---
```

```
331             function createText(svg, x, y, textContent, className) {
332                 const text = document.createElementNS('http://www.w3.org/2000/svg', 'text')
                        ;
333                 text.setAttribute('x', x);
334                 text.setAttribute('y', y);
335                 text.setAttribute('class', className);
336                 text.setAttribute('text-anchor', 'middle'); // Keep middle align for labels
337                 text.setAttribute('font-size', '12px');
338                 text.textContent = textContent;
339                 svg.appendChild(text);
340             }
341
342             function drawScaleBreakSymbol(svg, x, y) {
343                 const breakOffset = 4; // How far apart the lines are
344                 const breakHeight = 8; // How tall the zig-zag is
345                 const breakLine1 = document.createElementNS('http://www.w3.org/2000/svg', '
                        line');
346                 breakLine1.setAttribute('x1', x - breakOffset);
347                 breakLine1.setAttribute('y1', y - breakHeight);
348                 breakLine1.setAttribute('x2', x + breakOffset);
349                 breakLine1.setAttribute('y2', y + breakHeight);
350                 breakLine1.setAttribute('class', 'number-line-break');
351                 svg.appendChild(breakLine1);
352                  const breakLine2 = document.createElementNS('http://www.w3.org/2000/svg',
                        'line');
353                 breakLine2.setAttribute('x1', x + breakOffset); // Swapped x1/x2
354                 breakLine2.setAttribute('y1', y - breakHeight);
355                 breakLine2.setAttribute('x2', x - breakOffset); // Swapped x1/x2
356                 breakLine2.setAttribute('y2', y + breakHeight);
357                 breakLine2.setAttribute('class', 'number-line-break');
358                 svg.appendChild(breakLine2);
359             }
360
361             function createJumpArrow(svg, x1, y1, x2, y2, jumpArcHeight) {
362                 const path = document.createElementNS('http://www.w3.org/2000/svg', 'path')
                        ;
363                 const cx = (x1 + x2) / 2;
364                 const cy = y1 - jumpArcHeight; // Arc is above the line
365                 path.setAttribute('d', `M ${x1} ${y1} Q ${cx} ${cy} ${x2} ${y1}`);
366                 path.setAttribute('class', 'jump-arrow');
367                 svg.appendChild(path);
368
369                 // Arrowhead
370                 const jumpArrowHead = document.createElementNS('http://www.w3.org/2000/svg'
                        , 'path');
371                 const dx = x2 - cx; // Approx direction vector
372                 const dy = y1 - cy;
373                 const angleRad = Math.atan2(dy, dx);
374                 const angleDeg = angleRad * (180 / Math.PI);
375                 jumpArrowHead.setAttribute('class', 'jump-arrow-head');
376                 jumpArrowHead.setAttribute('d', `M 0 0 L ${arrowSize} ${arrowSize/2} L ${
                        arrowSize} ${-arrowSize/2} Z`);
377                 jumpArrowHead.setAttribute('transform', `translate(${x2}, ${y1}) rotate(${
                        angleDeg + 180})`);
```

```
            svg.appendChild(jumpArrowHead);
        }

        function drawStoppingPoint(svg, x, y, labelText) {
            const circle = document.createElementNS('http://www.w3.org/2000/svg', '
                circle');
            circle.setAttribute('cx', x);
            circle.setAttribute('cy', y);
            circle.setAttribute('r', 5);
            circle.setAttribute('class', 'stopping-point');
            svg.appendChild(circle);
            // Label below the point
            createText(svg, x, y + labelOffsetBase * 1.5, labelText, 'number-line-label
                ');
        }
    }

    function typesetMath() {
        // Placeholder
    }

});
</script>

</body>
</html>
```

References

Hackenberg, A. (2025). *Course notes* [Unpublished course notes].