Addition Strategies: Counting On By Bases and then Ones (COBO)

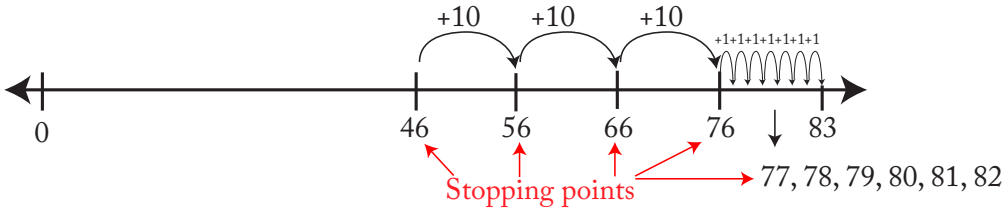
Compiled by: Theodore M. Savich

March 12, 2025

Transcript

Video from Carpenter et al. (1999). Strategy descriptions and examples adapted from Hackenberg (2025).

- **Teacher:** Max has 46 comic books. For his birthday, his father gives him 37 more comic books. How many comic books does Max have now?
- **Lauren:** Forty-six . . .
- **Teacher:** He gets 37 more for his birthday.
- **Lauren:** Ok. 46, 56, 66, 76, 77, 78, 79, 80, 81, 82, 83. It's 83.
- **Teacher:** Good work.



Notation Representing Sarah's Solution:

$$
\begin{aligned}
46 + 37 &= \square \\
46 + 10 &= 56 \\
56 + 10 &= 66 \\
66 + 10 &= 76 \\
76 + 1 &= 77 \\
77 + 1 &= 78 \\
78 + 1 &= 79 \\
79 + 1 &= 80 \\
80 + 1 &= 81 \\
81 + 1 &= 82 \\
82 + 1 &= 83
\end{aligned}
$$

Description of Strategy:

Objective: Description of Counting On by Bases and Then Ones (COBO) Begin with one of the numbers. Break the other number into its base units and its ones. Then, "count on" by adding each base unit one at a time, followed by each individual one.

Why are number lines useful for demonstrating this strategy? COBO is essentially a jump strategy—you start at one number and make "jumps" equal to the other number's base units, then add in the remaining ones. Number lines are ideal because they visually display jumps of varying lengths and directions. They serve as a picture of the process: a jump representing a full base is clearly larger (by a factor of the base) than a jump of a single unit.

Good number line illustrations should:

- Clearly represent the relative sizes of the jumps—each base jump should be exactly as many times larger than a single-unit jump as the base indicates, with all base jumps the same size and all one-unit jumps identical.
- Indicate the position of 0, or mark a break if that portion of the line isn't drawn to scale.
- Use arrows to indicate direction—when adding, the jumps go to the right (or upward); when subtracting, they go to the left (or downward).
- Mark all landing points clearly—the numbers you would speak aloud when counting on by bases and then ones, just as Lauren demonstrated.

Counting On by Bases and Then Ones (COBO)

Description of Strategy

- **Objective:** Start with one addend, add bases from the other addend one by one, then add ones one by one.
- **Example:** $46 + 37$
  - Start at 46.
  - Add tens one by one: $46 \to 56 \to 66 \to 76$.
  - Add ones one by one: $76 \to 77 \to \ldots \to 83$.

Automaton Type

Finite State Automaton (FSA) with Counters: Counters are used to manage the repeated addition:

- **BaseCounter:** Number of base units (e.g., tens) to add.
- **OneCounter:** Number of ones to add.
- **Sum:** The running total.

Formal Description of the Automaton

We define the automaton as the tuple

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

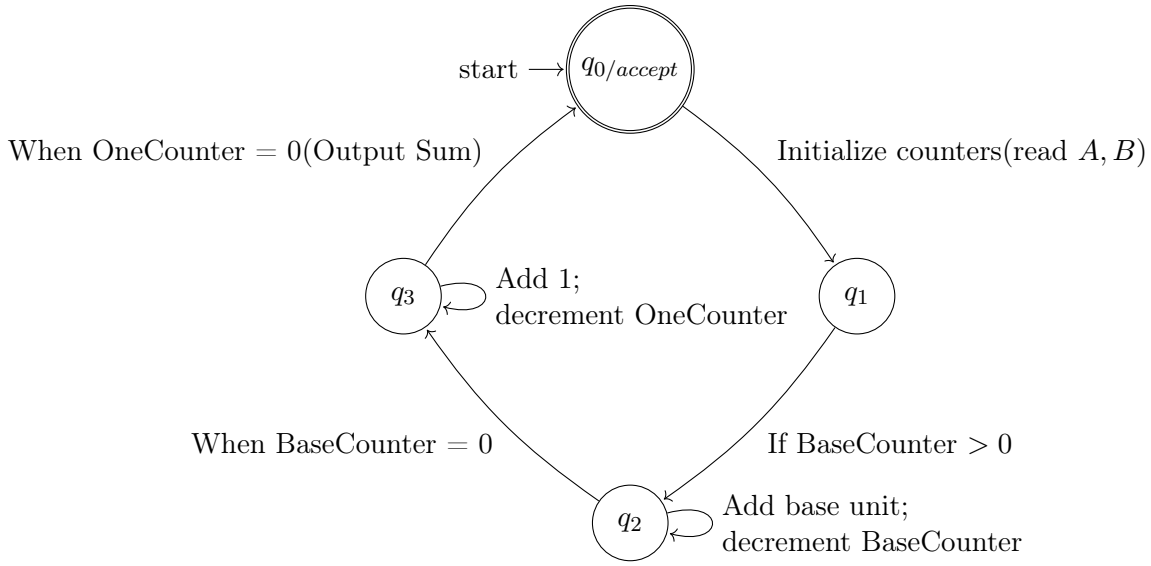
where:

- $Q = \{q_{0/accept}, q_1, q_2, q_3\}$ is the finite set of states. Here, the start state $q_{0/accept}$ is also the accept state.
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +\}$ is the input alphabet (suitable for representing the addends).
- $F = \{q_{0/accept}\}$ is the set of accepting states.
- $C = \{\text{BaseCounter}, \text{OneCounter}, \text{Sum}\}$ is the set of counters.
- δ is the transition function defined by:

  1. $\delta(q_{0/accept}, \text{“}A, B\text{”}) = (q_1, \text{ update: Sum} \leftarrow A, \text{ BaseCounter} \leftarrow \lfloor B/10 \rfloor, \text{ OneCounter} \leftarrow B \bmod 10)$
     (Read inputs A and B; initialize the Sum to A and set the counters based on B.)
  2. $\delta(q_1, \varepsilon) = (q_2, \text{ if BaseCounter} > 0)$
     (If there are base units to add, proceed to add them.)
  3. $\delta(q_2, \varepsilon) = (q_2, \text{ update: Sum} \leftarrow \text{Sum} + \text{baseUnit}, \text{ BaseCounter} \leftarrow \text{BaseCounter} - 1)$
     (In state q_2, repeatedly add one base unit (e.g., 10) to Sum while decrementing BaseCounter.)
  4. $\delta(q_2, \varepsilon) = (q_3, \text{ if BaseCounter} = 0)$
     (When no more base units remain, switch to adding ones.)
  5. $\delta(q_3, \varepsilon) = (q_3, \text{ update: Sum} \leftarrow \text{Sum} + 1, \text{ OneCounter} \leftarrow \text{OneCounter} - 1)$
     (In state q_3, repeatedly add 1 to Sum while decrementing OneCounter.)
  6. $\delta(q_3, \varepsilon) = (q_{0/accept}, \text{ if OneCounter} = 0)$
     (When OneCounter reaches 0, output the final Sum and return to the closed start/accept state.)

Automaton Diagram for COBO

The following diagram arranges the four states on a circle with $q_{0/accept}$ serving as both the start and accept state.


start
$q_{0/accept}$
When OneCounter = 0(Output Sum)
Initialize counters(read A, B)
q_3
Add 1;
decrement OneCounter
q_1
When BaseCounter = 0
If BaseCounter > 0
q_2
Add base unit;
decrement BaseCounter

HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Addition Strategies: Counting On By Bases and Ones (COBO)</title>
    <style>
        body {
            font-family: sans-serif;
        }
        #diagramCOBOSVG {
            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;
        }
        .number-line-label {
            font-size: 12px;
            text-anchor: end; /* Change from middle to start for right-alignment */
        }
        /* Thinner jump arrows */
        .jump-arrow {
            fill: none;
            stroke: blue;
            stroke-width: 1; /* thinner stroke */
        }
        .jump-arrow-head {
            fill: blue;
            stroke: blue;
        }
        .jump-label {
            font-size: 8px; /* Reduced from 12px to 8px */
            text-anchor: middle;
            fill: blue;
        }
        .tens-jump-label {
            font-size: 12px;
            text-anchor: middle;
            fill: blue;
        }
        .stopping-point {
            fill: red;
            stroke: black;
            stroke-width: 1;
        }
```

```
        /* New extended tick styles */
        .extended-tick {
            stroke: black;
            stroke-width: 1;
        }
        .extended-tick-label {
            font-size: 12px;
            text-anchor: start;
            fill: blue;
        }
        /* Number line arrowhead */
        .number-line-arrow {
            fill: black;
            stroke: black;
        }
    </style>
</head>
<body>

<h1>Addition Strategies: Counting On By Bases and Then Ones (COBO)</h1>

<div>
    <label for="coboAddend1">Addend 1:</label>
    <input type="number" id="coboAddend1" value="46">
</div>
<div>
    <label for="coboAddend2">Addend 2:</label>
    <input type="number" id="coboAddend2" value="37">
</div>

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

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

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

</svg>

<script>
document.addEventListener('DOMContentLoaded', function() {
    const outputElement = document.getElementById('coboOutput');
    const coboAddend1Input = document.getElementById('coboAddend1');
    const coboAddend2Input = document.getElementById('coboAddend2');
    const diagramCOBOSVG = document.getElementById('diagramCOBOSVG');

    window.runCOBOAutomaton = function() {
        const addend1 = parseInt(coboAddend1Input.value);
        const addend2 = parseInt(coboAddend2Input.value);
```

```
        if (isNaN(addend1) || isNaN(addend2)) {
            outputElement.textContent = 'Please enter valid numbers for both addends';
            return;
        }

        // Build text explanation
        let output = `<h2>Counting On by Bases and Ones (COBO)</h2>`;
        output += `<p><strong>Problem:</strong> ${addend1} + ${addend2}</p>`;

        const tens = Math.floor(addend2 / 10) * 10;
        const ones = addend2 % 10;

        output += `<p>Step 1: Split ${addend2} into ${tens} + ${ones}</p>`;

        let currentSum = addend1;
        const tensSteps = [];
        if (tens > 0) {
            output += `<p>Step 2: Count on by tens</p>`;
            for (let i = 10; i <= tens; i += 10) {
                tensSteps.push({ from: currentSum, to: currentSum + 10, action: 'Add 10' })
                    ;
                currentSum += 10;
            }
            tensSteps.forEach(step => {
                output += `<p>${step.from} + ${step.action} = ${step.to}</p>`;
            });
        }

        const onesSteps = [];
        if (ones > 0) {
            output += `<p>Step ${tens > 0 ? '3' : '2'}: Count on by ones</p>`;
            for (let i = 1; i <= ones; i++) {
                onesSteps.push({ from: currentSum, to: currentSum + 1, action: 'Add 1' });
                currentSum += 1;
            }
            onesSteps.forEach(step => {
                output += `<p>${step.from} + ${step.action} = ${step.to}</p>`;
            });
        }

        output += `<p>Result: ${addend1} + ${addend2} = ${currentSum}</p>`;
        outputElement.innerHTML = output;

        // Draw the diagram
        drawNumberLineDiagram(addend1, addend2, tensSteps, onesSteps, currentSum);
    };

    function drawNumberLineDiagram(addend1, addend2, tensSteps, onesSteps, finalSum) {
        const svg = diagramCOBOSVG;
        svg.innerHTML = ''; // Clear any previous diagram

        // Dimensions
        const width = parseFloat(svg.getAttribute('width'));
        const height = parseFloat(svg.getAttribute('height'));
```

```
        const marginLeft = 50;
        const marginRight = 50;
        const numberLineY = height / 2;

        // Arc heights
        const TENS_ARC_HEIGHT = 30;
        const ONES_ARC_HEIGHT = 15;

        // We'll place 0 at x=marginLeft, then a scale break if addend1 > ~0
        // then line from addend1 to finalSum to scale
        const zeroX = marginLeft;
        const breakX = zeroX + 15;
        const lineStartX = breakX + 15;

        // Extend the line 10 points past the final sum
        const extendAmount = 10;
        const numericRange = Math.max(finalSum + extendAmount - addend1, 1); // at least 1 to avoid /0
        const lineEndX = width - marginRight;

        // Calculate scale after considering the extension
        const scale = (lineEndX - lineStartX) / numericRange;

        // Draw the number line from zero to the break point
        drawLine(zeroX, numberLineY, lineStartX, numberLineY);


        // Draw "0" tick
        drawTick(zeroX, numberLineY, 10);
        createText(zeroX, numberLineY + 15, '0');

        // If addend1 > 0, draw scale break
        if (addend1 > 0) {
            drawScaleBreakSymbol(breakX, numberLineY);
        }

        // Main line with arrowhead
        drawLine(lineStartX, numberLineY, lineEndX, numberLineY);

        // Add arrowhead to the right end of number line
        const arrowSize = 10;
        const arrowHead = document.createElementNS('http://www.w3.org/2000/svg', 'path');
        arrowHead.setAttribute('d', `M ${lineEndX-arrowSize} ${numberLineY-arrowSize/2} L ${lineEndX} ${numberLineY} L ${lineEndX-arrowSize} ${numberLineY+arrowSize/2} Z`);
        arrowHead.setAttribute('class', 'number-line-arrow');
        svg.appendChild(arrowHead);

        // Convert a value to x-coord
        function valueToX(v) {
            return lineStartX + (v - addend1) * scale;
        }

        // Mark addend1
```

```
        drawTick(valueToX(addend1), numberLineY, 10);
        createText(valueToX(addend1), numberLineY + 15, addend1.toString());

        // Draw tens jumps
        tensSteps.forEach((step) => {
            const x1 = valueToX(step.from);
            const x2 = valueToX(step.to);
            createJumpArrow(svg, x1, numberLineY, x2, numberLineY, TENS_ARC_HEIGHT);
            // Mark landing
            drawTick(x2, numberLineY, 10);
            createText(x2, numberLineY + 15, step.to.toString());

            // Add "+10" label above the arc
            const midX = (x1 + x2) / 2;
            const labelY = numberLineY - TENS_ARC_HEIGHT - 5;
            const txtTensLabel = document.createElementNS('http://www.w3.org/2000/svg', '
                text');
            txtTensLabel.setAttribute('x', midX);
            txtTensLabel.setAttribute('y', labelY);
            txtTensLabel.setAttribute('class', 'tens-jump-label');
            txtTensLabel.textContent = '+10';
            svg.appendChild(txtTensLabel);
        });

        // Draw ones jumps
        onesSteps.forEach((step, index) => {
            const x1 = valueToX(step.from);
            const x2 = valueToX(step.to);

            // Create jump arrow as before
            createJumpArrow(svg, x1, numberLineY, x2, numberLineY, ONES_ARC_HEIGHT);

            // Create extended tick with increasing length based on index
            const tickLength = 10 + (index * 10); // Increase by 10px for each subsequent
                tick
            const extendedTick = document.createElementNS('http://www.w3.org/2000/svg', '
                line');
            extendedTick.setAttribute('x1', x2);
            extendedTick.setAttribute('y1', numberLineY);
            extendedTick.setAttribute('x2', x2);
            extendedTick.setAttribute('y2', numberLineY + tickLength); // Now going down
                instead of up
            extendedTick.setAttribute('class', 'extended-tick');
            svg.appendChild(extendedTick);

            // Add the number label at the end of the tick mark
            createText(x2, numberLineY + tickLength + 15, step.to.toString());

            // Add "+1" label above the arc
            const midX = (x1 + x2) / 2;
            const labelY = numberLineY - ONES_ARC_HEIGHT - 5;
            const txtOneLabel = document.createElementNS('http://www.w3.org/2000/svg', '
                text');
            txtOneLabel.setAttribute('x', midX);
```

```
            txtOneLabel.setAttribute('y', labelY);
            txtOneLabel.setAttribute('class', 'jump-label');
            txtOneLabel.textContent = '+1';
            svg.appendChild(txtOneLabel);
        });

        // Mark finalSum if not already marked in ones steps
        if (onesSteps.length === 0 || onesSteps[onesSteps.length - 1].to !== finalSum) {
            drawTick(valueToX(finalSum), numberLineY, 10);
            createText(valueToX(finalSum), numberLineY + 15, finalSum.toString());
        }

        // ----------------- Drawing Helpers ------------------

        function drawLine(x1, y1, x2, y2) {
            const line = document.createElementNS('http://www.w3.org/2000/svg', 'line');
            line.setAttribute('x1', x1);
            line.setAttribute('y1', y1);
            line.setAttribute('x2', x2);
            line.setAttribute('y2', y2);
            line.setAttribute('class', 'number-line-tick');
            svg.appendChild(line);
        }

        function drawTick(x, y, size) {
            const tick = document.createElementNS('http://www.w3.org/2000/svg', 'line');
            tick.setAttribute('x1', x);
            tick.setAttribute('y1', y - size / 2);
            tick.setAttribute('x2', x);
            tick.setAttribute('y2', y + size / 2);
            tick.setAttribute('class', 'number-line-tick');
            svg.appendChild(tick);
        }

        function createText(x, y, textContent) {
            const txt = document.createElementNS('http://www.w3.org/2000/svg', 'text');
            txt.setAttribute('x', x);
            txt.setAttribute('y', y);
            txt.setAttribute('class', 'number-line-label');
            txt.textContent = textContent;
            svg.appendChild(txt);
        }

        function drawScaleBreakSymbol(x, y) {
            // Two small diagonal lines crossing
            const breakLine1 = document.createElementNS('http://www.w3.org/2000/svg', '
                line');
            breakLine1.setAttribute('x1', x);
            breakLine1.setAttribute('y1', y - 8);
            breakLine1.setAttribute('x2', x + 8);
            breakLine1.setAttribute('y2', y + 8);
            breakLine1.setAttribute('class', 'number-line-break');
            svg.appendChild(breakLine1);

```

```
            const breakLine2 = document.createElementNS('http://www.w3.org/2000/svg', 'line');
            breakLine2.setAttribute('x1', x);
            breakLine2.setAttribute('y1', y + 8);
            breakLine2.setAttribute('x2', x + 8);
            breakLine2.setAttribute('y2', y - 8);
            breakLine2.setAttribute('class', 'number-line-break');
            svg.appendChild(breakLine2);
        }

        /**
         * Draws a curved jump (quadratic Bezier) from (x1, y1) to (x2, y2),
         * with control point arcHeight above the line, and attaches a manual arrowhead.
         */
        function createJumpArrow(svg, x1, y1, x2, y2, jumpArcHeight) {
            // Quadratic Bezier arc
            const cx = (x1 + x2) / 2; // midpoint in x
            const cy = y1 - jumpArcHeight; // arc above the line

            // Main arc path
            const arcPath = document.createElementNS('http://www.w3.org/2000/svg', 'path');
            arcPath.setAttribute('d', `M ${x1} ${y1} Q ${cx} ${cy} ${x2} ${y2}`);
            arcPath.setAttribute('class', 'jump-arrow');
            svg.appendChild(arcPath);

            // Compute angle for arrowhead
            // derivative at end of Q-bezier ~ direction from control point to end
            const dx = x2 - cx;
            const dy = y2 - cy;
            const angleRad = Math.atan2(dy, dx);
            const angleDeg = angleRad * (180 / Math.PI);

            // Manual arrowhead as small filled triangle
            const arrowSize = 5;
            const arrowHead = document.createElementNS('http://www.w3.org/2000/svg', 'path');
            arrowHead.setAttribute('class', 'jump-arrow-head');
            arrowHead.setAttribute('d', `M 0 0 L ${arrowSize} ${arrowSize/2} L ${arrowSize} ${-arrowSize/2} Z`);
            arrowHead.setAttribute('transform', `translate(${x2}, ${y2}) rotate(${angleDeg + 180})`);
            svg.appendChild(arrowHead);
        }
    };
});
</script>

</body>
</html>
```

References

Carpenter, T. P., Fennema, E., Franke, M. L., Levi, L., & Empson, S. B. (1999). Children's mathematics: Cognitively guided instruction – videotape logs [supplementary material]. In *Children's mathematics: Cognitively guided instruction*. Heinemann, in association with The National Council of Teachers of Mathematics, Inc.

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