Addition Strategies: Rearranging to Make Bases (RMB)

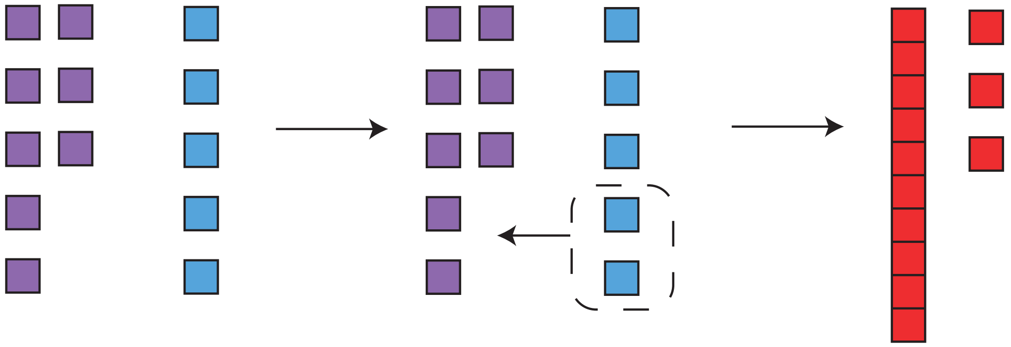
Compiled by: Theodore M. Savich

April 1, 2025

Transcript

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

- **Teacher:** Lucy is eight fish. She buys five more fish. How many fish will Lucy have then?
- **Sarah:** 13.
- **Teacher:** How'd you get 13?
- **Sarah:** Well, because eight plus two is ten, but then two plus three is five. And she wants to buy five more fish. So you take care of two, and you need to add three more. And so I add three more, and you get 13.



Notation Representing Sarah's Solution:

$$\begin{aligned}
&8 + 5 = \square \\
&8 + 2 = 10 \\
&2 + 3 = 5 \\
&8 + 5 = 10 + 3 \\
&8 + 5 = 13
\end{aligned}$$

Description of Strategy:

Objective: Rearranging to Make Bases (RMB) means shifting the extra ones from one addend over to the other so that one of the numbers becomes a complete multiple of the base (a whole "group" of that base). This rearrangement simplifies the addition process because there are established patterns for adding an exact multiple of the base. In other words, when you add a full group of base units to a number, the ones digit stays the same while only the digit representing the base (like the tens place) increases.

Rearranging to Make Bases (RMB)

Description of Strategy

- **Objective:** Make one of the addends a whole number of bases by moving ones from the other addend.
- **Example:** $8 + 5$
  - Move 2 ones from 5 to 8 to make 10.
  - Remaining ones in the second addend: $5 - 2 = 3$.
  - Add the adjusted numbers: $10 + 3 = 13$.

Automaton Type

Pushdown Automaton (PDA): Needed to handle digits and to remember the number of ones moved via the stack.

Formal Description of the Automaton

We define the PDA as the 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_{0/accept}, Z_0, F)$$

where

- $Q = \{q_{0/accept},\, q_1,\, q_2,\, q_3,\, q_4,\, q_5\}$ is the finite set of states.
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +\}$ is the input alphabet (suitable for representing addends).
- $\Gamma = \{Z_0\} \cup \{x \mid x \in \mathbb{N}\}$ is the stack alphabet, where:
  - Z_0 is the initial (bottom) stack symbol.
  - A symbol x represents the number of ones moved.
- $q_{0/accept}$ is the start state, which is also the accept state.
- Z_0 is the initial stack symbol.
- $F = \{q_{0/accept}\}$ is the set of accepting states.

The transition function

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$$

is defined by the following key transitions:

1. $\delta\Big(q_{0/accept},\, \text{“}A, B\text{”},\, Z_0\Big) = \{(q_1,\, Z_0)\}$ (Read inputs A and B).
2. $\delta(q_1,\, \varepsilon,\, Z_0) = \{(q_2,\, Z_0)\}$ (Determine the ones digits of A and B).

3. $\delta(q_2,\, \varepsilon,\, Z_0) = \{(q_3,\, Z_0)\}$ (Compute the number of ones needed to make A a full base).

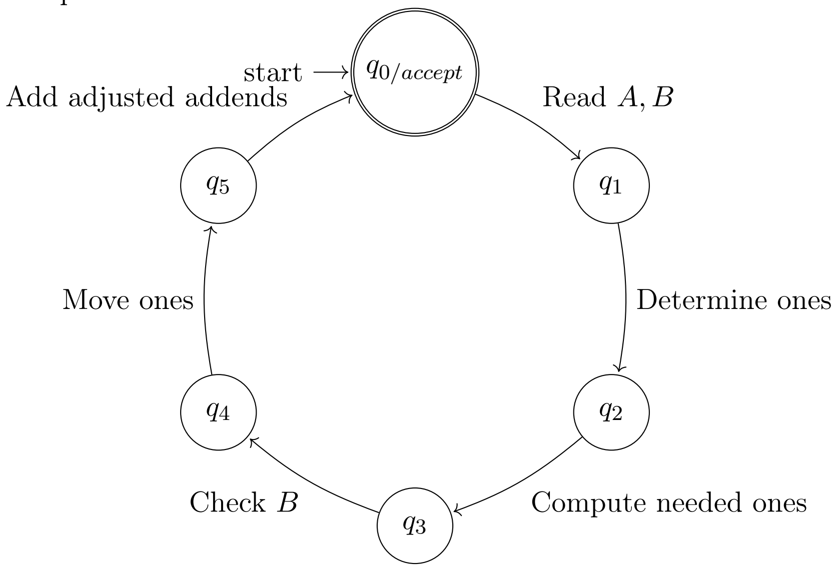
4. $\delta(q_3,\, \varepsilon,\, Z_0) = \{(q_4,\, k\, Z_0)\}$ (If B has at least k ones, push k onto the stack).

5. $\delta(q_4,\, \varepsilon,\, k) = \{(q_5,\, k)\}$ (Move k ones from B to A and adjust the addends).

6. $\delta(q_5,\, \varepsilon,\, k) = \{(q_{0/accept},\, Z_0)\}$ (Add the adjusted numbers, output the result, and pop k from the stack).

Automaton Diagram for RMB

The following TikZ picture arranges the 6 states on a circle, with $q_{0/accept}$ serving as both the start and accept state.



HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Rearranging to Make Bases (RMB) Addition</title>
    <style>
        body { font-family: sans-serif; }
        #diagramRMBSVG { border: 1px solid #d3d3d3; } /* Style SVG like canvas */
        #outputContainer { margin-top: 20px; }
        .diagram-label { font-size: 14px; display: block; margin-bottom: 5px; } /*
            Improved label styling */
    </style>
</head>
<body>

    <h1>Addition Strategies: Rearranging to Make Bases (RMB)</h1>

    <div>
        <label for="addend1">Addend 1:</label>
        <input type="number" id="addend1" value="18">
    </div>
    <div>
        <label for="addend2">Addend 2:</label>
        <input type="number" id="addend2" value="15">
    </div>

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

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

    <h2>Diagram:</h2>
    <svg id="diagramRMBSVG" width="600" height="700"></svg> <!-- Increased height -->

    <!-- New button for viewing PDF documentation -->
    <button onclick="openPdfViewer()">Want to learn more about this strategy? Click here.</button>

    <script>
document.addEventListener(’DOMContentLoaded’, function() {
    const rmbOutputElement = document.getElementById(’rmbOutput’);
    const rmbAddend1Input = document.getElementById(’addend1’);
    const rmbAddend2Input = document.getElementById(’addend2’);
    const diagramRMBSVG = document.getElementById(’diagramRMBSVG’);

    if (!rmbOutputElement || !diagramRMBSVG) {
        console.warn("Element␣rmbOutput␣or␣diagramRMBSVG␣not␣found");
        return;
    }

    window.runRMBAutomaton = function() {
        try {
            const addend1 = parseInt(rmbAddend1Input.value);
            const addend2 = parseInt(rmbAddend2Input.value);

            if (isNaN(addend1) || isNaN(addend2)) {
                rmbOutputElement.textContent = "Please␣enter␣valid␣numbers␣for␣both␣addends";
                return;
            }

            let output = ’’;
            output += ‘<h2>Rearranging to Make Bases (RMB)</h2><br><br>‘;
            output += ‘<p><strong>Problem:</strong> ${addend1} + ${addend2}</p><br><br>‘;

            const toMakeBase = (10 - (addend1 % 10)) % 10;

            // Strategy variables
            let newAddend1, newAddend2, result, transferAmount;
            let fromFirst = false; // Whether we’re transferring from addend1 to addend2

            // Case 1: When addend1 is already a multiple of 10
            if (toMakeBase === 0) {
                // Instead of direct calculation, decompose addend2 into tens and ones
                const a2_tens = Math.floor(addend2 / 10);
                const a2_ones = addend2 % 10;

                output += ‘${addend1} is already a multiple of 10.<br>‘;
                output += ‘Step 1: Break down ${addend2} into tens and ones<br>‘;
                output += ‘ ${addend2} = ${a2_tens * 10} + ${a2_ones}<br><br>‘;
                output += ‘Step 2: Add the parts to ${addend1}<br>‘;
                output += ‘ ${addend1} + ${a2_tens * 10} = ${addend1 + a2_tens * 10}<br>‘;
                output += ‘ ${addend1 + a2_tens * 10} + ${a2_ones} = ${addend1 + addend2}<br><br>‘;
                output += ‘Result: ${addend1} + ${addend2} = ${addend1 + addend2}‘;

                newAddend1 = addend1;
                newAddend2 = addend2;
                transferAmount = 0;
                result = addend1 + addend2;

                rmbOutputElement.innerHTML = output;
                drawRMBDiagram(’diagramRMBSVG’, addend1, addend2, transferAmount, newAddend1, newAddend2, result, fromFirst);
                return;
            }

            // Case 2: When addend2 is too small to provide needed units
            if (addend2 < toMakeBase) {
                // Instead of direct calculation, transfer from addend1 to complete addend2 to a base
                fromFirst = true;
                const a1_ones = addend1 % 10;
                const toCompleteAddend2 = 10 - addend2;

                // We’ll move units from addend1 to addend2
                transferAmount = Math.min(a1_ones, toCompleteAddend2);
                newAddend1 = addend1 - transferAmount;
                newAddend2 = addend2 + transferAmount;
                result = newAddend1 + newAddend2; // Will equal addend1 + addend2

                output += ‘${addend2} is too small to provide the ${toMakeBase} units needed for ${addend1}.<br>‘;
                output += ‘Step 1: Move ${transferAmount} from ${addend1} to ${addend2}<br>‘;
                output += ‘ ${addend1} - ${transferAmount} = ${newAddend1}<br>‘;
                output += ‘ ${addend2} + ${transferAmount} = ${newAddend2}<br><br>‘;

                // If we made a complete base in addend2
                if (newAddend2 % 10 === 0) {
                    output += ‘Step 2: Now ${newAddend2} is a complete base (multiple of 10)<br>‘;
                } else {
                    output += ‘Step 2: Even after moving, we can’t␣make␣a␣complete␣base,␣but␣we␣rearranged␣for␣easier␣mental␣addition<br>‘;
                }

                output += `Step 3: Add the rearranged numbers<br>`;
                output += `${newAddend1} + ${newAddend2} = ${result}<br><br>`;
                output += `Result: ${addend1} + ${addend2} = ${result}`;

                rmbOutputElement.innerHTML = output;
                drawRMBDiagram('diagramRMBSVG', addend1, addend2, transferAmount, newAddend1, newAddend2, result, fromFirst);
                return;
            }

            // Original case: Standard RMB strategy
            transferAmount = toMakeBase;
            newAddend1 = addend1 + transferAmount;
            newAddend2 = addend2 - transferAmount;
            result = newAddend1 + newAddend2;

            output += `Step 1: Move ${transferAmount} from ${addend2} to ${addend1}<br>`;
            output += ` ${addend1} + ${transferAmount} = ${newAddend1} (now a multiple of 10)<br>`;
            output += ` ${addend2} - ${transferAmount} = ${newAddend2}<br><br>`;
            output += `Step 2: Add the rearranged numbers<br>`;
            output += `${newAddend1} + ${newAddend2} = ${result}<br><br>`;
            output += `Result: ${addend1} + ${addend2} = ${result}`;

            rmbOutputElement.innerHTML = output;

            // Draw RMB Diagram
            drawRMBDiagram('diagramRMBSVG', addend1, addend2, transferAmount, newAddend1, newAddend2, result, fromFirst);
        } catch (error) {
            rmbOutputElement.textContent = `Error: ${error.message}`;
        }
    };


    function drawRMBDiagram(svgId, addend1, addend2, transferAmount, newAddend1, newAddend2, result, fromFirst = false) {
        const svg = document.getElementById(svgId);
        if (!svg) return;
        svg.innerHTML = ''; // Clear SVG

        const svgWidth = parseFloat(svg.getAttribute('width'));
        const svgHeight = parseFloat(svg.getAttribute('height'));
        const blockUnitSize = 15; // Size of individual unit block
        const tenBlockWidth = blockUnitSize; // Width of 10-block rectangle
        const tenBlockHeight = blockUnitSize * 10; // Height of 10-block rectangle
        const blockSpacing = 5;
        const sectionSpacingY = 120; // Vertical spacing between sections
        const startX = 50;
        let currentY = 50;
        const colorAddend1 = 'purple';
        const colorAddend2 = 'blue';
        const colorBase = 'red';
        const colorTransfer = 'orange';

        // --- Original Addends (Horizontal Layout) ---
        createText(svg, startX, currentY, `Original Addends: ${addend1} + ${addend2}`); // Label
        currentY += 30; // Space after label

        // Draw Addend 1 (purple) on left
        let addend1X = startX;
        const a1_tens = Math.floor(addend1 / 10);
        const a1_ones = addend1 % 10;
        for (let i = 0; i < a1_tens; i++) {
            drawTenBlock(svg, addend1X, currentY, tenBlockWidth, tenBlockHeight, colorAddend1);
            addend1X += tenBlockWidth + blockSpacing;
        }
        let a1_onesX = addend1X;
        let movedFromFirstBlockPositions = [];
        for (let i = 0; i < a1_ones; i++) {
            const isTransferBlock = fromFirst && i >= a1_ones - transferAmount;
            const blockColor = isTransferBlock ? colorTransfer : colorAddend1;
            const blockY = currentY + i*(blockUnitSize + blockSpacing);
            drawBlock(svg, a1_onesX, blockY, blockUnitSize, blockUnitSize, blockColor);

            if (isTransferBlock) {
                movedFromFirstBlockPositions.push({
                    x: a1_onesX + blockUnitSize/2,
                    y: blockY + blockUnitSize/2
                });
            }
        }
        const addend1Width = (a1_tens > 0 ? (a1_tens*(tenBlockWidth + blockSpacing)) : 0) + (a1_ones > 0 ? blockUnitSize : 0);

        // Draw Addend 2 (blue) to the right of Addend 1
        let addend2X = startX + addend1Width + 50; // 50px horizontal spacing between addend groups
        const a2_tens = Math.floor(addend2 / 10);
        const a2_ones = addend2 % 10;
        for (let i = 0; i < a2_tens; i++) {
            drawTenBlock(svg, addend2X, currentY, tenBlockWidth, tenBlockHeight, colorAddend2);
            addend2X += tenBlockWidth + blockSpacing;
        }
        const addend2OnesX = addend2X;
        let movedFromSecondBlockPositions = [];
        for (let i = 0; i < a2_ones; i++) {
            const isTransferBlock = !fromFirst && i < transferAmount;
            const blockColor = isTransferBlock ? colorTransfer : colorAddend2;
            const blockY = currentY + i*(blockUnitSize + blockSpacing);
            drawBlock(svg, addend2OnesX, blockY, blockUnitSize, blockUnitSize, blockColor);

            if (isTransferBlock) {
                movedFromSecondBlockPositions.push({
                    x: addend2OnesX + blockUnitSize/2,
                    y: blockY + blockUnitSize/2
                });
            }
        }
        currentY += tenBlockHeight + sectionSpacingY; // Move down for the rearranged addends section

        // --- Rearranged Addends ---
        createText(svg, startX, currentY, `Rearranged to Make Base: ${newAddend1} + ${newAddend2}`); // Label
        currentY += 30; // Space after label

        // Draw Rearranged Addend 1 Blocks
        let currentX_newAddend1 = startX;
        const newAddend1_tens = Math.floor(newAddend1 / 10);
        const newAddend1_ones = newAddend1 % 10;

        // First draw tens
        let tensPositions = [];
        for (let i = 0; i < newAddend1_tens; i++) {
            const useColorBase = !fromFirst && newAddend1_tens > a1_tens && i === newAddend1_tens - 1;
            const blockColor = useColorBase ? colorBase : colorAddend1;
            drawTenBlock(svg, currentX_newAddend1, currentY, tenBlockWidth, tenBlockHeight, blockColor);

            if (useColorBase) {
                tensPositions.push({
                    x: currentX_newAddend1 + tenBlockWidth/2,
                    y: currentY + tenBlockHeight/2
                });
            }

            currentX_newAddend1 += tenBlockWidth + blockSpacing;
        }

        // Then draw ones
        for (let i = 0; i < newAddend1_ones; i++) {
            drawBlock(svg, currentX_newAddend1, currentY + i*(blockUnitSize + blockSpacing),
                     blockUnitSize, blockUnitSize, colorAddend1);
        }

        // Draw Rearranged Addend 2 Blocks
        const newAddend2_tens = Math.floor(newAddend2 / 10);
        const newAddend2_ones = newAddend2 % 10;
        let currentX_newAddend2 = currentX_newAddend1 + 40 + (newAddend1_ones > 0 ? blockUnitSize : 0); // Spacing after newAddend1

        // Draw tens
        for (let i = 0; i < newAddend2_tens; i++) {
            const useColorBase = fromFirst && newAddend2_tens > a2_tens && i === newAddend2_tens - 1;
            const blockColor = useColorBase ? colorBase : colorAddend2;
            drawTenBlock(svg, currentX_newAddend2, currentY, tenBlockWidth, tenBlockHeight, blockColor);

            if (useColorBase) {
                tensPositions.push({
                    x: currentX_newAddend2 + tenBlockWidth/2,
                    y: currentY + tenBlockHeight/2
                });
            }

            currentX_newAddend2 += tenBlockWidth + blockSpacing;
        }

        // Draw ones
        let onesPositions = [];
        for (let i = 0; i < newAddend2_ones; i++) {
            const isTransferredBlock = fromFirst && i >= a2_ones;
            const blockColor = isTransferredBlock ? colorTransfer : colorAddend2;
            const blockY = currentY + i*(blockUnitSize + blockSpacing);
            drawBlock(svg, currentX_newAddend2, blockY, blockUnitSize, blockUnitSize, blockColor);

            if (isTransferredBlock) {
                onesPositions.push({
                    x: currentX_newAddend2 + blockUnitSize/2,
                    y: blockY + blockUnitSize/2
                });
            }
        }

        // --- Draw Arrows Based on Strategy ---
        if (transferAmount > 0) {
            if (fromFirst) {
                // Case 2: Draw arrows from addend1 to addend2
                for (let i = 0; i < Math.min(movedFromFirstBlockPositions.length, onesPositions.length); i++) {
                    const start = movedFromFirstBlockPositions[i];
                    const end = onesPositions[i];
                    const controlX = (start.x + end.x) / 2;
                    const controlY = Math.min(start.y, end.y) - 40; // Control point above both
                    createCurvedArrow(svg, start.x, start.y, end.x, end.y, controlX, controlY);
                }

                // If we formed a full ten, draw arrow to tens block
                if (newAddend2 % 10 === 0 && tensPositions.length > 0) {
                    const start = movedFromFirstBlockPositions[0];
                    const end = tensPositions[0];
                    createText(svg, end.x + 15, end.y - 20, `Formed a base (10)`);
                }
```

```
␣␣␣␣␣␣␣␣␣␣␣␣}␣else␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣Standard␣case:␣Draw␣arrows␣from␣addend2␣to␣addend1
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣(let␣i␣=␣0;␣i␣<␣Math.min(movedFromSecondBlockPositions.length,␣tensPositions.length);␣i++)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣start␣=␣movedFromSecondBlockPositions[i];
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣end␣=␣tensPositions[i];
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣controlX␣=␣(start.x␣+␣end.x)␣/␣2;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣controlY␣=␣Math.min(start.y,␣end.y)␣-␣40;␣//␣Control␣point␣above␣both
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣createCurvedArrow(svg,␣start.x,␣start.y,␣end.x,␣end.y,␣controlX,␣controlY);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣createText(svg,␣tensPositions[0]?.x␣+␣15␣||␣startX␣+␣100,␣tensPositions[0]?.y␣-␣20␣||␣currentY␣-␣20,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣‘${transferAmount}␣moved␣to␣form␣base␣(10)‘);
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣}␣else␣if␣(addend1␣%␣10␣===␣0)␣{
␣␣␣␣␣␣␣␣␣␣␣␣//␣Case␣1:␣Already␣a␣multiple␣of␣10,␣show␣the␣decomposition
␣␣␣␣␣␣␣␣␣␣␣␣const␣a2_tens␣=␣Math.floor(addend2␣/␣10);
␣␣␣␣␣␣␣␣␣␣␣␣if␣(a2_tens␣>␣0)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣createText(svg,␣startX␣+␣80,␣currentY␣-␣40,␣‘Break␣down␣${addend2}␣=␣${a2_tens␣*␣10}␣+␣${addend2␣%␣10}‘);
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣}

␣␣␣␣␣␣␣␣//␣---␣Helper␣SVG␣drawing␣functions␣---
␣␣␣␣␣␣␣␣function␣drawBlock(svg,␣x,␣y,␣width,␣height,␣fill)␣{
␣␣␣␣␣␣␣␣␣␣␣␣const␣rect␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’rect’);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’x’,␣x);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’y’,␣y);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’width’,␣width);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’height’,␣height);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’fill’,␣fill);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’stroke’,␣’black’);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’stroke-width’,␣’1’);
␣␣␣␣␣␣␣␣␣␣␣␣svg.appendChild(rect);
␣␣␣␣␣␣␣␣}

␣␣␣␣␣␣␣␣function␣drawTenBlock(svg,␣x,␣y,␣width,␣height,␣fill)␣{
␣␣␣␣␣␣␣␣␣␣␣␣const␣group␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’g’);␣//␣Group␣for␣10-block
␣␣␣␣␣␣␣␣␣␣␣␣const␣backgroundRect␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’rect’);
␣␣␣␣␣␣␣␣␣␣␣␣backgroundRect.setAttribute(’x’,␣x);
␣␣␣␣␣␣␣␣␣␣␣␣backgroundRect.setAttribute(’y’,␣y);
␣␣␣␣␣␣␣␣␣␣␣␣backgroundRect.setAttribute(’width’,␣width);
␣␣␣␣␣␣␣␣␣␣␣␣backgroundRect.setAttribute(’height’,␣height);
␣␣␣␣␣␣␣␣␣␣␣␣backgroundRect.setAttribute(’fill’,␣fill);
␣␣␣␣␣␣␣␣␣␣␣␣backgroundRect.setAttribute(’stroke’,␣’black’);
␣␣␣␣␣␣␣␣␣␣␣␣backgroundRect.setAttribute(’stroke-width’,␣’1’);
␣␣␣␣␣␣␣␣␣␣␣␣group.appendChild(backgroundRect);

␣␣␣␣␣␣␣␣␣␣␣␣//␣Draw␣10␣unit␣blocks␣inside␣-␣vertical␣column
␣␣␣␣␣␣␣␣␣␣␣␣for␣(let␣i␣=␣0;␣i␣<␣10;␣i++)␣{
```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣unitBlock␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’rect’);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣unitBlock.setAttribute(’x’,␣x␣);␣//␣Same␣x␣for␣vertical␣column
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣unitBlock.setAttribute(’y’,␣y␣+␣i␣*␣blockUnitSize);␣//␣Stacked␣vertically
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣unitBlock.setAttribute(’width’,␣blockUnitSize);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣unitBlock.setAttribute(’height’,␣blockUnitSize);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣unitBlock.setAttribute(’fill’,␣fill);␣//␣Same␣fill␣as␣outer␣rect
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣unitBlock.setAttribute(’stroke’,␣’lightgrey’);␣//␣Lighter␣border␣for␣units
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣unitBlock.setAttribute(’stroke-width’,␣’0.5’);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣group.appendChild(unitBlock);
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣svg.appendChild(group);
␣␣␣␣␣␣␣␣}

␣␣␣␣␣␣␣␣function␣drawGroupRect(svg,␣x,␣y,␣width,␣height)␣{
␣␣␣␣␣␣␣␣␣␣␣␣const␣rect␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’rect’);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’x’,␣x);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’y’,␣y);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’width’,␣width);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’height’,␣height);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’fill’,␣’none’);␣//␣No␣fill␣for␣group␣rect
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’stroke’,␣’black’);
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’stroke-dasharray’,␣’5 5’);␣//␣Dashed␣border␣for␣grouping
␣␣␣␣␣␣␣␣␣␣␣␣rect.setAttribute(’stroke-width’,␣’1’);
␣␣␣␣␣␣␣␣␣␣␣␣svg.appendChild(rect);
␣␣␣␣␣␣␣␣}

␣␣␣␣␣␣␣␣function␣createText(svg,␣x,␣y,␣textContent)␣{
␣␣␣␣␣␣␣␣␣␣␣␣const␣text␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’text’);
␣␣␣␣␣␣␣␣␣␣␣␣text.setAttribute(’x’,␣x);
␣␣␣␣␣␣␣␣␣␣␣␣text.setAttribute(’y’,␣y);
␣␣␣␣␣␣␣␣␣␣␣␣text.setAttribute(’class’,␣’diagram-label’);
␣␣␣␣␣␣␣␣␣␣␣␣text.setAttribute(’text-anchor’,␣’start’);
␣␣␣␣␣␣␣␣␣␣␣␣text.setAttribute(’font-size’,␣’14px’);
␣␣␣␣␣␣␣␣␣␣␣␣text.textContent␣=␣textContent;
␣␣␣␣␣␣␣␣␣␣␣␣svg.appendChild(text);
␣␣␣␣␣␣␣␣}

␣␣␣␣␣␣␣␣function␣createCurvedArrow(svg,␣x1,␣y1,␣x2,␣y2,␣cx,␣cy)␣{
␣␣␣␣␣␣␣␣␣␣␣␣const␣path␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’path’);
␣␣␣␣␣␣␣␣␣␣␣␣path.setAttribute(’d’,␣‘M␣${x1}␣${y1}␣Q␣${cx}␣${cy}␣${x2}␣${y2}‘);
␣␣␣␣␣␣␣␣␣␣␣␣path.setAttribute(’fill’,␣’none’);
␣␣␣␣␣␣␣␣␣␣␣␣path.setAttribute(’stroke’,␣’black’);
␣␣␣␣␣␣␣␣␣␣␣␣path.setAttribute(’stroke-width’,␣’2’);
␣␣␣␣␣␣␣␣␣␣␣␣svg.appendChild(path);

␣␣␣␣␣␣␣␣␣␣␣␣␣//␣Arrowhead
␣␣␣␣␣␣␣␣␣␣␣␣const␣arrowHead␣=␣document.createElementNS("http://www.w3.org/2000/svg",␣’path’);
␣␣␣␣␣␣␣␣␣␣␣␣const␣arrowSize␣=␣5;
␣␣␣␣␣␣␣␣␣␣␣␣arrowHead.setAttribute(’d’,␣‘M␣${x2}␣${y2}␣L␣${x2␣-␣arrowSize}␣${y2␣-␣arrowSize}␣L␣${x2␣+␣arrowSize}␣${y2␣-␣arrowSize}␣Z‘);
␣␣␣␣␣␣␣␣␣␣␣␣arrowHead.setAttribute(’fill’,␣’black’);
```

```
␣␣␣␣␣␣␣␣␣␣␣␣svg.appendChild(arrowHead);
␣␣␣␣␣␣␣␣}
␣␣␣␣}

␣␣␣␣//␣New␣function␣for␣opening␣PDF␣documentation
␣␣␣␣function␣openPdfViewer()␣{
␣␣␣␣␣␣␣␣//␣Opens␣the␣PDF␣documentation␣for␣the␣strategy.
␣␣␣␣␣␣␣␣window.open(’../SAR_ADD_RMB.pdf’,␣’_blank’);
␣␣␣␣}

});
␣␣␣␣</script>

</body>
</html>
```

References

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

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