Subtraction Strategies: Chunking

Compiled by: Theodore M. Savich

March 28, 2025

Transcript

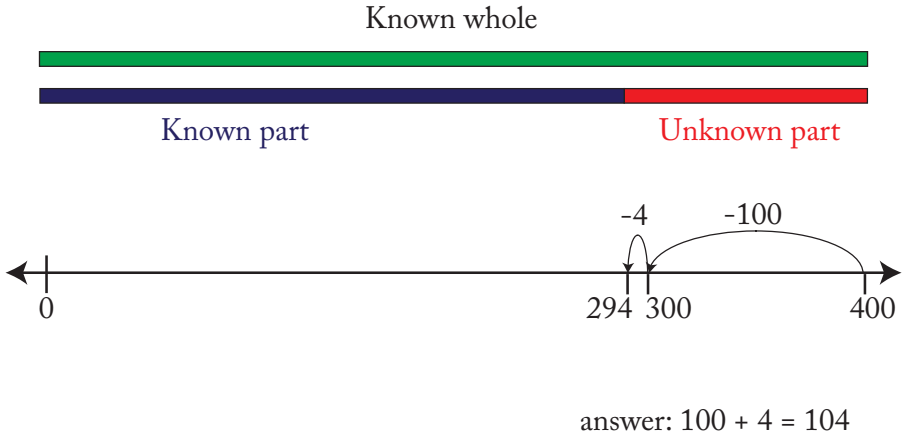
Video from Carpenter et al. (1999). Strategy descriptions and examples adapted from Hackenberg (2025).

- **Teacher:**"One summer T.J. saved $400. At the end of the summer she spent $294 on a new bike. How much money did T.J. have then?"
- **Student:** "400 takeaway 200 is 200. I just put the 4 on the side right now. So then 200 takeaway 90 is 110. So then 110 takeaway 4 is 106.
- **Teacher:** "So how much money did she have left?"
- **Student:** "106."
- **Teacher:** "Nice job."

Here is the notation below to show what the student did:

$$\begin{aligned} 400 - 200 &= 200 \\ 200 - 90 &= 110 \\ 110 - 4 &= 106 \end{aligned}$$

c.) Chunk back from the known whole to the known part



answer: 100 + 4 = 104

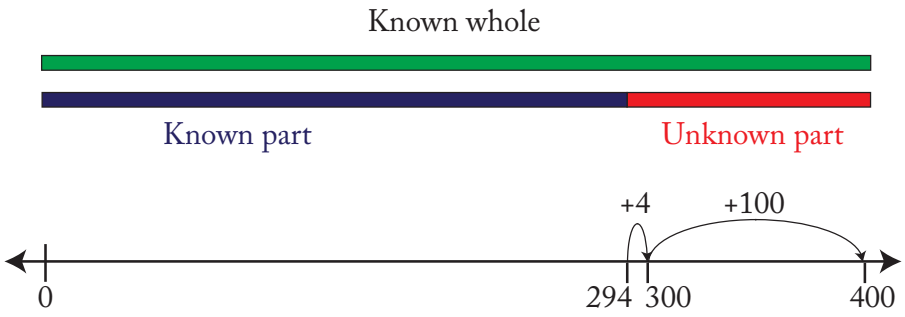
However, this is only one of three structurally different ways that chunking can show up in subtraction.

(a) **Chunking backwards (by known part)** The student starts at the known whole and subtracts backwards by the known part. They arrive at the unknown part.

(b) **Chunking forwards** The student subtracts the known whole from the known part.

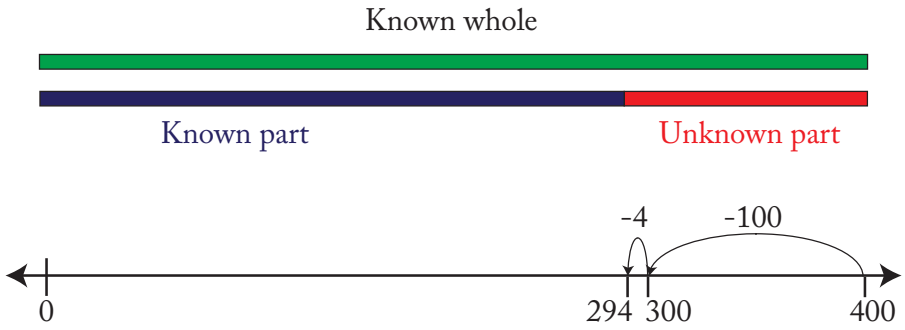
(c) **Chunking backwards (to the known part)** The student starts at the known whole and subtracts backwards until they reach the known part.

b.) Chunking forwards (from known part to known whole)



answer: 4 + 100 = 104

c.) Chunk back from the known whole to the known part



answer: 100 + 4 = 104

Description of Strategy

- Subtract the subtrahend (known part) from the minuend (known whole) by breaking the subtrahend into bases and ones and subtracting in strategic chunks.
- Start from the subtrahend and add strategic chunks to reach the minuend, summing the chunks to find the difference.
- Start at the minuend and subtract strategic chunks until you reach the subtrahend, summing the chunks to find the difference.

Automaton Type - only one type of chunking is analyzed here

Finite State Automaton (FSA) with Counters: Counters are used to manage the sequential subtraction:

- **BaseCounter:** Counts the number of base chunks to subtract.
- **OneCounter:** Counts the number of ones to subtract.
- **Difference:** Accumulates the running difference.

Formal Description of the Automaton

We define the automaton as the tuple

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

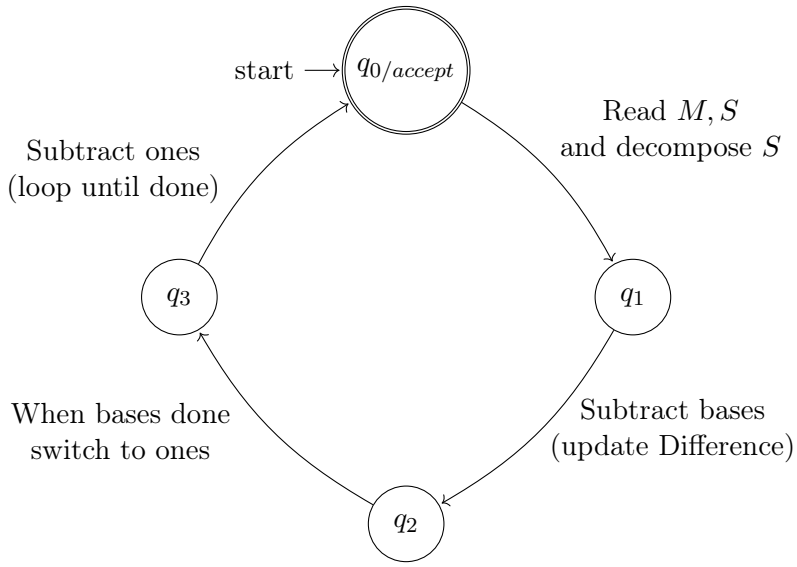
where:

- $Q = \{q_{0/accept}, q_1, q_2, q_3\}$ is the set of states.
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is the input alphabet (representing the digits of the minuend M and subtrahend S).
- $q_{0/accept}$ is the start state, which is also the accept state.
- $F = \{q_{0/accept}\}$ is the set of accepting states.
- $C = \{\text{BaseCounter, OneCounter, Difference}\}$ is the set of counters.

The transition function δ is defined as follows:

1. $\delta(q_{0/accept}, \text{“}M, S\text{”}) =$
   $(q_1, \text{initialize: set Difference} \leftarrow M, \text{ Decompose } S \text{ into BaseCounter and OneCounter}).$
2. $\delta(q_1, \varepsilon) =$
   $(q_2, \text{while BaseCounter} > 0 : \text{ Difference} \leftarrow \text{Difference} - (\text{base chunk}), \text{ decrement BaseCounter}).$
3. $\delta(q_2, \varepsilon) =$
   $(q_3, \text{when BaseCounter} = 0).$
4. $\delta(q_3, \varepsilon) =$
   $(q_3, \text{while OneCounter} > 0 : \text{ Difference} \leftarrow \text{Difference} - (\text{ones chunk}), \text{ decrement OneCounter}).$
5. $\delta(q_3, \varepsilon) =$
   $(q_{0/accept}, \text{when OneCounter} = 0 : \text{ output Difference}).$

Automaton Diagram for Chunking by Bases and Ones (Forwards or Backwards)



HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Subtraction Strategies: Chunking</title>
    <style>
        body { font-family: sans-serif; }
        #diagramSubChunkingSVG { border: 1px solid #d3d3d3; }
        #outputContainer { margin-top: 20px; }
        fieldset { margin: 15px 0; border: 1px solid #ccc; padding: 10px;}
        legend { font-weight: bold; }
        label { margin-right: 10px; }
        /* Number line styles */
        .number-line-tick { stroke: black; stroke-width: 1; }
        .number-line-break { stroke: black; stroke-width: 1; } /* Solid for zig-zag */
        .number-line-label { font-size: 12px; text-anchor: middle; }
        .jump-arrow {
            fill: none; /* <-- Ensure no fill for the arc */
            stroke-width: 2;
         }
        .jump-arrow-head {
            stroke-width: 2;
            fill: none; /* <-- Ensure no fill for the arrow head */
         }
        .jump-label { font-size: 12px; text-anchor: middle; }
        .stopping-point { fill: red; stroke: black; stroke-width: 1; }
        .number-line-arrow { fill: black; stroke: black;}
        /* Colors for strategies */
        .strategy-a { stroke: darkred; } /* Backwards by part */
        .strategy-b { stroke: darkgreen; } /* Forwards */
        .strategy-c { stroke: darkblue; } /* Backwards to part */
    </style>
</head>
<body>

<h1>Subtraction Strategies: Chunking</h1>

<div>
    <label for="chunkMinuend">Minuend (Whole):</label>
    <input type="number" id="chunkMinuend" value="400">
</div>
<div>
    <label for="chunkSubtrahend">Subtrahend (Part):</label>
    <input type="number" id="chunkSubtrahend" value="294">
</div>

<fieldset>
    <legend>Choose Chunking Strategy:</legend>
    <input type="radio" id="strategyA" name="chunkingStrategy" value="A" checked>
    <label for="strategyA">A: Backwards (by Known Part)</label><br>
    <input type="radio" id="strategyB" name="chunkingStrategy" value="B">
    <label for="strategyB">B: Forwards (from Known Part)</label><br>
    <input type="radio" id="strategyC" name="chunkingStrategy" value="C">
```

```
    <label for="strategyC">C: Backwards (to Known Part)</label><br>
</fieldset>


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

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

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

<script>
document.addEventListener('DOMContentLoaded', function() {
    const outputElement = document.getElementById('subChunkingOutput');
    const minuendInput = document.getElementById('chunkMinuend');
    const subtrahendInput = document.getElementById('chunkSubtrahend');
    const diagramSVG = document.getElementById('diagramSubChunkingSVG');
    const strategyRadios = document.getElementsByName('chunkingStrategy');

    // --- All Helper SVG Drawing Functions Defined Here --- (Keep from previous version)
        ---
    function createText(svg, x, y, textContent, className = 'number-line-label') {
        const text = document.createElementNS("http://www.w3.org/2000/svg", 'text');
        text.setAttribute('x', x);
        text.setAttribute('y', y);
        text.setAttribute('class', className);
        text.setAttribute('text-anchor', 'middle');
        text.textContent = textContent;
        svg.appendChild(text);
    }

    function drawTick(svg, 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 drawScaleBreakSymbol(svg, x, y) {
        const breakOffset = 4;
        const breakHeight = 8;
        const breakLine1 = document.createElementNS('http://www.w3.org/2000/svg', 'line');
        breakLine1.setAttribute('x1', x - breakOffset);
        breakLine1.setAttribute('y1', y - breakHeight);
        breakLine1.setAttribute('x2', x + breakOffset);
        breakLine1.setAttribute('y2', y + breakHeight);
```

```
        breakLine1.setAttribute('class', 'number-line-break');
        svg.appendChild(breakLine1);
        const breakLine2 = document.createElementNS('http://www.w3.org/2000/svg', 'line');
        breakLine2.setAttribute('x1', x + breakOffset);
        breakLine2.setAttribute('y1', y - breakHeight);
        breakLine2.setAttribute('x2', x - breakOffset);
        breakLine2.setAttribute('y2', y + breakHeight);
        breakLine2.setAttribute('class', 'number-line-break');
        svg.appendChild(breakLine2);
    }

    function createJumpArrow(svg, x1, y1, x2, y2, jumpArcHeight, direction = 'forward',
        colorClass = 'strategy-b', arrowSize = 5) {
        const path = document.createElementNS('http://www.w3.org/2000/svg', 'path');
        const cx = (x1 + x2) / 2;
        const cy = y1 - jumpArcHeight;
        path.setAttribute('d', `M ${x1} ${y1} Q ${cx} ${cy} ${x2} ${y1}`);
        path.setAttribute('class', `jump-arrow ${colorClass}`); // Apply strategy color to
            arc stroke
        path.setAttribute('fill', 'none'); // Explicitly set fill to none to prevent
            filling
        svg.appendChild(path);

        const arrowHead = document.createElementNS('http://www.w3.org/2000/svg', 'path');
        const dx = x2 - cx;
        const dy = y1 - cy;
        const angleRad = Math.atan2(dy, dx);
        let angleDeg = angleRad * (180 / Math.PI);
        arrowHead.setAttribute('class', `jump-arrow-head ${colorClass}`); // Apply
            strategy color to head fill/stroke

        if (direction === 'forward') {
            angleDeg += 180;
            arrowHead.setAttribute('d', `M 0 0 L ${arrowSize} ${arrowSize/2} L ${arrowSize
                } ${-arrowSize/2} Z`);
        } else { // backward
            arrowHead.setAttribute('d', `M 0 0 L ${-arrowSize} ${arrowSize/2} L ${-
                arrowSize} ${-arrowSize/2} Z`);
        }
        arrowHead.setAttribute('transform', `translate(${x2}, ${y1}) rotate(${angleDeg})`)
            ;
        svg.appendChild(arrowHead);
    }


     function drawStoppingPoint(svg, x, y, labelText, labelOffsetBase) {
         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);
         createText(svg, x, y + labelOffsetBase * 1.5, labelText, 'number-line-label');
     }
```

```
153     // --- End Helper Functions ---
154
155     // --- Main Automaton Function ---
156     window.runSubtractionChunkingAutomaton = function() {
157         try {
158             const minuend = parseInt(minuendInput.value); // M (Whole)
159             const subtrahend = parseInt(subtrahendInput.value); // S (Known Part)
160             let selectedStrategy = 'A'; // Default
161             for (const radio of strategyRadios) {
162                 if (radio.checked) {
163                     selectedStrategy = radio.value;
164                     break;
165                 }
166             }
167
168             if (isNaN(minuend) || isNaN(subtrahend)) {
169                 outputElement.textContent = 'Please␣enter␣valid␣numbers␣for␣Minuend␣and␣
                        Subtrahend';
170                 diagramSVG.innerHTML = '';
171                 return;
172             }
173             if (subtrahend > minuend && selectedStrategy !== 'B') {
174                  outputElement.textContent = 'Subtrahend␣cannot␣be␣greater␣than␣Minuend␣for
                         ␣strategies␣A␣and␣C.';
175                  diagramSVG.innerHTML = '';
176                  return;
177             }
178
179
180             let output = `<h2>Subtraction Chunking (Strategy ${selectedStrategy})</h2>\n\n
                    `;
181             output += `<p><strong>Problem:</strong> ${minuend} - ${subtrahend}</p>\n\n`;
182
183             const chunkSteps = [];
184             let finalDifference = 0;
185             let currentVal = 0;
186             let targetVal = 0;
187             let direction = 'backward'; // Default for A and C
188             let startPoint = minuend;
189             let endPoint = 0; // Will be calculated
190             let totalChunkSum = 0; // For strategies B and C
191
192              let stepCounter = 1; // Initialize step counter
193
194
195             // --- Logic based on Selected Strategy ---
196             switch (selectedStrategy) {
197                 //=========================================
198                 case 'A': // Chunking Backwards (by Known Part) M - S = ?
199                 //=========================================
200                     output += `Strategy A: Start at ${minuend}, subtract chunks of ${
                            subtrahend}.\n`;
201                     currentVal = minuend;
202                     targetVal = minuend - subtrahend;
```

```
                startPoint = minuend;

                let tensToSubtract = Math.floor(subtrahend / 10) * 10;
                let onesToSubtract = subtrahend % 10;


                // Subtract Tens Chunk
                if (tensToSubtract > 0) {
                    output += `<p>Step ${stepCounter++}: Subtract tens chunk</p>\n`;
                    chunkSteps.push({ from: currentVal, to: currentVal - tensToSubtract
                        , label: `-${tensToSubtract}` });
                    output += `<p>${currentVal} - ${tensToSubtract} = ${currentVal -
                        tensToSubtract}</p>\n`;
                    currentVal -= tensToSubtract;
                }

                // Subtract Ones Chunks Strategically
                if (onesToSubtract > 0) {
                    output += `<p>Step ${stepCounter++}: Subtract ones chunk(s)</p>\n`;
                    while (onesToSubtract > 0) {
                        let onesToPreviousTen = currentVal % 10;
                        if (onesToPreviousTen === 0 && onesToSubtract > 0)
                            onesToPreviousTen = 10;

                        let chunk = Math.min(onesToSubtract, onesToPreviousTen);
                         if (chunk === 0 && onesToSubtract > 0) chunk = onesToSubtract;
                         if (chunk === 0) break;

                         chunkSteps.push({ from: currentVal, to: currentVal - chunk,
                             label: `-${chunk}` });
                         output += `<p>${currentVal} - ${chunk} = ${currentVal - chunk
                             }`;
                         if (chunk === onesToPreviousTen && chunk !== onesToSubtract &&
                             (currentVal - chunk) % 10 === 0) output += ` (Making
                             previous ten)`;
                         output += `</p>\n`;
                         currentVal -= chunk;
                         onesToSubtract -= chunk;
                    }
                }

                finalDifference = currentVal;
                endPoint = finalDifference;
                output += `\n<p><strong>Result (Final Position):</strong> ${
                    finalDifference}</p>`;
                break;

            //====================================================
            case 'B': // Chunking Forwards (from Known Part) S + ? = M
            //====================================================
                output += `Strategy B: Start at ${subtrahend}, add chunks to reach ${
                    minuend}.\n`;
                currentVal = subtrahend;
                targetVal = minuend;
```

```
                startPoint = subtrahend;
                endPoint = minuend;
                direction = 'forward';
                totalChunkSum = 0;

                while (currentVal < targetVal) {
                    output += `<p>Step ${stepCounter++}: Add chunk</p>\n`;
                    let diff = targetVal - currentVal;
                    let chunk = 0;
                    let explanation = '';

                    let onesToNextTen = (10 - (currentVal % 10)) % 10;
                    if (onesToNextTen > 0 && onesToNextTen <= diff) {
                        chunk = onesToNextTen;
                        explanation = '(Making the next ten)';
                    } else {
                        let tensToNextHundred = (100 - (currentVal % 100)) % 100;
                        if (currentVal % 10 === 0 && tensToNextHundred > 0 &&
                            tensToNextHundred <= diff) {
                            chunk = tensToNextHundred;
                            explanation = '(Making the next hundred)';
                        } else {
                            if (diff >= 100) chunk = Math.floor(diff / 100) * 100;
                            else if (diff >= 10) chunk = Math.floor(diff / 10) * 10;
                            else chunk = diff;
                        }
                    }
                    if (chunk <= 0) { chunk = diff; explanation = ''; };

                    chunkSteps.push({ from: currentVal, to: currentVal + chunk, label:
                        `+${chunk}` });
                    output += `<p>${currentVal} + ${chunk} = ${currentVal + chunk} ${
                        explanation}</p>\n`;
                    currentVal += chunk;
                    totalChunkSum += chunk;
                }

                finalDifference = totalChunkSum;
                 output += `\n<p><strong>Result (Sum of Chunks):</strong> ${
                    finalDifference}</p>`;
                break;

            //=====================================================
            case 'C': // Chunking Backwards (to Known Part) M - ? = S (REVISED LOGIC)
            //=====================================================
                output += `Strategy C: Start at ${minuend}, subtract chunks to reach ${
                    subtrahend}.\n`;
                currentVal = minuend;
                targetVal = subtrahend;
                startPoint = minuend;
                endPoint = subtrahend;
                direction = 'backward';
                totalChunkSum = 0;

```

```
297                 while (currentVal > targetVal) {
298                     output += `<p>Step ${stepCounter++}: Subtract chunk</p>\n`;
299                     let diff = currentVal - targetVal;
300                     let chunk = 0;
301                     let explanation = '';
302
303                     // Priority 1: Subtract ones chunk to land on a ten?
304                     let onesToPreviousTen = currentVal % 10;
305                     // Only do this if it doesn't overshoot the target AND makes sense
306                     if (onesToPreviousTen > 0 && onesToPreviousTen <= diff) {
307                         chunk = onesToPreviousTen;
308                         explanation = '(Making␣previous␣ten)';
309                     } else {
310                         // Priority 2: Subtract tens chunk to land on a hundred?
311                         let tensToPreviousHundred = currentVal % 100;
312                          // Only do this if at a multiple of 10, it doesn't overshoot,
                                and makes sense
313                         if (currentVal % 10 === 0 && tensToPreviousHundred > 0 &&
                                tensToPreviousHundred <= diff) {
314                             chunk = tensToPreviousHundred;
315                             explanation = '(Making␣previous␣hundred)';
316                         } else {
317                             // Priority 3: Subtract largest power of 10 chunk possible
                                    without overshooting
318                             if (diff >= 100) {
319                                 chunk = Math.floor(diff / 100) * 100; // Largest
                                        hundreds chunk <= diff
320                             } else if (diff >= 10) {
321                                 chunk = Math.floor(diff / 10) * 10; // Largest tens
                                        chunk <= diff
322                             } else {
323                                 chunk = diff; // Subtract remaining ones if < 10
324                             }
325                         }
326                     }
327
328                     // Final check to ensure chunk doesn't overshoot
329                     chunk = Math.min(chunk, diff);
330                     // Ensure positive chunk if difference exists
331                     if (chunk <= 0 && diff > 0) { chunk = diff; explanation = ''; };
332
333                     if (chunk === 0) break; // Safety exit if no chunk calculated
334
335                     chunkSteps.push({ from: currentVal, to: currentVal - chunk, label:
                            `-${chunk}` });
336                     output += `<p>${currentVal} - ${chunk} = ${currentVal - chunk} ${
                            explanation}</p>\n`;
337                     currentVal -= chunk;
338                     totalChunkSum += chunk;
339                 }
340
341                  finalDifference = totalChunkSum;
342                  output += `\n<p><strong>Result (Sum of Chunks):</strong> ${
                         finalDifference}</p>`;
```

```
                break;
            //=====================================================
        }


        outputElement.innerHTML = output;
        typesetMath();

        // --- Draw Number Line Diagram ---
        let allValues = [startPoint, endPoint];
        chunkSteps.forEach(step => { allValues.push(step.from); allValues.push(step.to
            ); });
        let diagramMin = Math.min(...allValues);
        let diagramMax = Math.max(...allValues);

        drawNumberLineDiagram(diagramSVG,
            startPoint, endPoint,
            diagramMin, diagramMax,
            chunkSteps, direction, selectedStrategy);


    } catch (error) {
        console.error("Error in runSubtractionChunkingAutomaton:", error);
        outputElement.textContent = `Error: ${error.message}`;
    }
};

function drawNumberLineDiagram(svg, startValue, endValue, diagramMin, diagramMax,
    chunkSteps, direction, strategy) {
    if (!svg || typeof svg.setAttribute !== 'function') { console.error("Invalid SVG
        element..."); 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;
    const tickHeight = 10;
    const labelOffsetBase = 20;
    const jumpHeightLarge = 60;
    const jumpHeightSmall = 40;
    const jumpLabelOffset = 15;
    const arrowSize = 5;
    const scaleBreakThreshold = 40;

    // Calculate scale and handle potential break
    let displayRangeStart = diagramMin;
    let scaleStartX = startX;
    let drawScaleBreak = false;

    if (diagramMin > scaleBreakThreshold) {
        displayRangeStart = diagramMin - 10;
        scaleStartX = startX + 30;
```

```
            drawScaleBreak = true;
            drawScaleBreakSymbol(svg, scaleStartX - 15, numberLineY);
            drawTick(svg, startX, numberLineY, tickHeight);
            createText(svg, startX, numberLineY + labelOffsetBase, '0', 'number-line-label');
        } else {
            displayRangeStart = 0;
            drawTick(svg, startX, numberLineY, tickHeight);
            createText(svg, startX, numberLineY + labelOffsetBase, '0', 'number-line-label');
        }

        const displayRangeEnd = diagramMax + 10;
        const displayRange = Math.max(displayRangeEnd - displayRangeStart, 1);
        const scale = (endX - scaleStartX) / displayRange;

        // Function to convert value to X coordinate
        function valueToX(value) {
            if (value < displayRangeStart && drawScaleBreak) { return scaleStartX - 10; }
            const scaledValue = scaleStartX + (value - displayRangeStart) * scale;
            return Math.max(scaleStartX, Math.min(scaledValue, endX));
        }

        // Draw the main visible segment of the number line
         const mainLineStartX = valueToX(displayRangeStart);
         const mainLineEndX = valueToX(displayRangeEnd);
         const numberLine = document.createElementNS('http://www.w3.org/2000/svg', 'line');
         numberLine.setAttribute('x1', mainLineStartX);
         numberLine.setAttribute('y1', numberLineY);
         numberLine.setAttribute('x2', mainLineEndX);
         numberLine.setAttribute('y2', numberLineY);
         numberLine.setAttribute('class', 'number-line-tick');
         svg.appendChild(numberLine);

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


        // Draw Ticks and Labels
        function drawTickAndLabel(value, index) {
            const x = valueToX(value);
            if (x < scaleStartX - 5 && value !== 0) return;

            drawTick(svg, x, numberLineY, tickHeight); // Pass svg
            const labelOffset = labelOffsetBase * (index % 2 === 0 ? 1 : -1.5); // Stagger
            createText(svg, x, numberLineY + labelOffset, value.toString(), 'number-line-label'); // Pass svg
```

```
        }

        // Draw ticks for start, end, and all intermediate points
        let allPoints = new Set([startValue, endValue, ...chunkSteps.map(s => s.to), ...
            chunkSteps.map(s => s.from)]);
        let sortedPoints = Array.from(allPoints).sort((a, b) => a - b);
        let pointIndexMap = {};
        let currentIndex = 0;
        sortedPoints.forEach(point => {
            if (point >= displayRangeStart || (point === 0 && !drawScaleBreak)) {
                if (!(point < displayRangeStart && drawScaleBreak)){
                    pointIndexMap[point] = currentIndex++;
                    drawTickAndLabel(point, pointIndexMap[point]);
                }
            }
        });


        // Draw chunk jumps
        let strategyColorClass = `strategy-${strategy.toLowerCase()}`;
        chunkSteps.forEach((step, index) => {
            const x1 = valueToX(step.from);
            const x2 = valueToX(step.to);
            if (x1 > endX || x2 > endX || x1 < scaleStartX || x2 < scaleStartX || x1 ==
                x2 ) return;

            const isLargeChunk = Math.abs(step.to - step.from) >= 10;
            const currentJumpHeight = isLargeChunk ? jumpHeightLarge : jumpHeightSmall;
            const staggerOffset = index % 2 === 0 ? 0 : currentJumpHeight * 0.4;

            createJumpArrow(svg, x1, numberLineY, x2, numberLineY, currentJumpHeight +
                staggerOffset, direction, strategyColorClass, arrowSize); // Pass
                arrowSize
            createText(svg, (x1 + x2) / 2, numberLineY - (currentJumpHeight +
                staggerOffset) - jumpLabelOffset, step.label, `jump-label ${
                strategyColorClass}`);
        });

        // Start point marker
        if (valueToX(startValue) >= scaleStartX) {
            drawStoppingPoint(svg, valueToX(startValue), numberLineY, 'Start',
                labelOffsetBase); // Pass labelOffsetBase
        }
    }

    function typesetMath() { /* Placeholder */ }

    // Initial run on page load
    runSubtractionChunkingAutomaton();

});
</script>

</body>
```

```
488 </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].