Strategic Multiplicative Reasoning: Conversion to Bases and Ones (CBO)

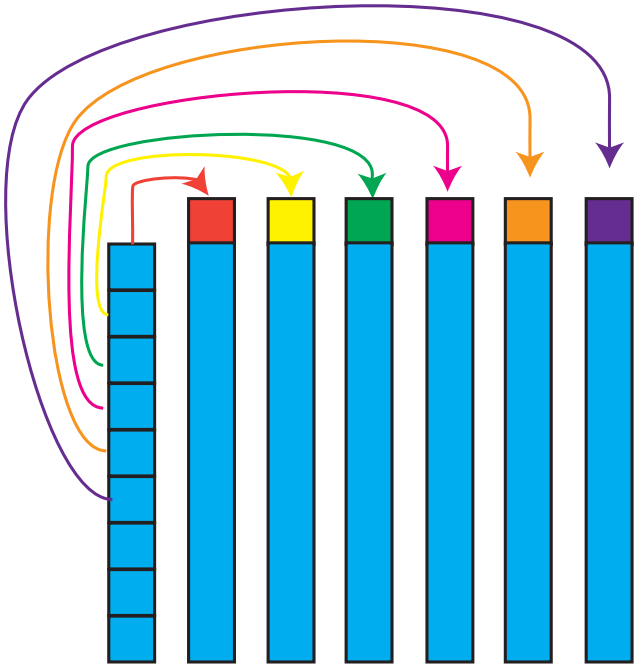
Compiled by: Theodore M. Savich

March 30, 2025

Transcript

Strategy descriptions and examples adapted from Hackenberg (2025).

- **Teacher:** You have 7 mini cans of soda. Each can has 9 ounces of soda in it. How many ounces of soda do you have total?
- **George:** Well, you could take one of the 9 ounces and put an extra ounce into all other cans. That would give you 6 tens with 3 ounces leftover. So, 63.
- **Teacher:** Great!



$$\begin{aligned}\text{Seven} \times 9 &= \text{Six} \times 9 + 9 \\ &= \text{Six} \times 9 + 6 + 3 \\ &= \text{Six} \times (9 + 1) + 3 \\ &= \text{Six} \times 10 + 3 \\ &= 63\end{aligned}$$

Begin with groups of a known size. The objective is to form groups that equal the base size. To achieve this, break one group apart and redistribute its individual units to other groups until they form complete bases; repeat with additional groups if necessary. Typically, some units will remain ungrouped. The total count is then the sum of the complete bases and any leftover units.

Conversion to Bases and Ones (CBO)

Description of Strategy:

- **Objective:** Rearrange the items from groups to make complete base units by combining ones from different groups.
- **Method:** Break apart groups and redistribute ones to form full base units (e.g., tens).

Automaton Type:

Pushdown Automaton (PDA): The stack is used to represent the redistribution of ones in order to form complete base units.

Formal Description of the Automaton

We define the PDA as the 7-tuple

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

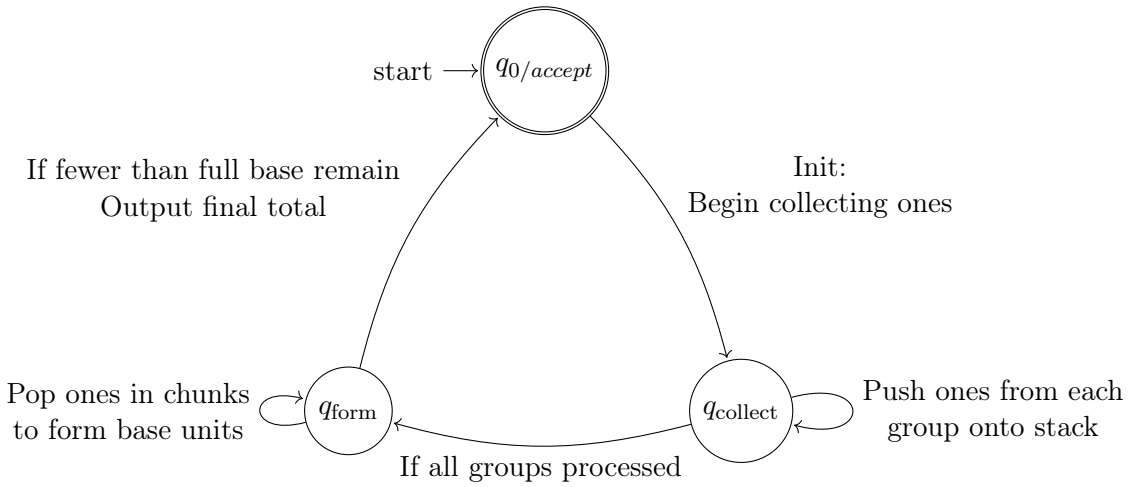
where:

- $Q = \{q_{0/accept}, q_{\text{collect}}, q_{\text{form}}\}$ is the set of states. Here, $q_{0/accept}$ serves as both the start and accept state.
- Σ is the input alphabet (encoding the group information, e.g., number of groups and ones per group).
- $\Gamma = \{Z_0\} \cup \{1\}$ is the stack alphabet, where Z_0 is the initial stack symbol and the symbol 1 represents a single one.
- $q_{0/accept}$ 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 by:

1. $\delta(q_{0/accept}, \text{“init”}, Z_0) = \{(q_{\text{collect}}, Z_0)\}$
   (Initialize the process to collect ones from the groups.)
2. In state q_{collect}: $\delta(q_{\text{collect}}, \varepsilon, x) = \{(q_{\text{collect}}, 1x)\}$ for any $x \in \Gamma$
   (For each group, push the ones (e.g., S ones) onto the stack.)
   Additionally, when all groups have been processed (i.e. a designated input symbol signals that the count of groups equals N), we have: $\delta(q_{\text{collect}}, \varepsilon, Z_0) = \{(q_{\text{form}}, Z_0)\}$.
3. In state q_{form}: $\delta(q_{\text{form}}, \varepsilon, 1) = \{(q_{\text{form}}, \varepsilon)\}$ (simulate popping a one) repeated until fewer than $BSize$ symbols remain on the stack. When fewer than $BSize$ ones remain (i.e., a full base unit cannot be formed), $\delta(q_{\text{form}}, \varepsilon, Z_0) = \{(q_{0/accept}, Z_0)\}$
   (Output the final result, which is implicitly represented by the distribution of ones on the stack.)

Automaton Diagram for Conversion to Bases and Ones



HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Multiplication: Conversion to Bases and Ones (CBO - Redistribution)</title>
    <style>
        body { font-family: sans-serif; }
        #cboDiagram { border: 1px solid #d3d3d3; min-height: 500px; }
        #outputContainer { margin-top: 20px; }
        .diagram-label { font-size: 14px; display: block; margin-bottom: 10px; font-weight
            : bold;}
        .notation-line { margin: 0.2em 0; margin-left: 1em; font-family: monospace;}
        .notation-line.problem { font-weight: bold; margin-left: 0;}
        /* Block Styles */
        .block { stroke: black; stroke-width: 0.5; }
        .ten-block-bg { stroke: black; stroke-width: 1; }
        .hundred-block-bg { stroke: black; stroke-width: 1; }
        .unit-block-inner { stroke: lightgrey; stroke-width: 0.5; }
        .initial-group-item { fill: teal; } /* Color for items in initial groups */
        .final-ten { fill: lightgreen; } /* Color for final ten blocks */
        .final-one { fill: gold; } /* Color for final one blocks */
        .redistribute-arrow { /* Style for arrows showing redistribution */
            fill: none;
            stroke: orange;
            stroke-width: 1.5;
            stroke-dasharray: 4 2;
        }
         .redistribute-arrow-head {
            fill: orange;
            stroke: orange;
        }

    </style>
</head>
<body>

<h1>Strategic Multiplicative Reasoning: Conversion to Bases and Ones (CBO -
    Redistribution)</h1>

<div>
    <label for="cboGroups">Number of Groups (N):</label>
    <input type="number" id="cboGroups" value="7" min="1"> <!-- George's␣Example␣-->
</div>
<div>
␣␣␣␣<label␣for="cboItems">Items␣per␣Group␣(S):</label>
␣␣␣␣<input␣type="number"␣id="cboItems"␣value="9"␣min="1">␣<!--␣George's Example -->
</div>

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

<div id="outputContainer">
    <h2>Explanation (Notation):</h2>
    <div id="cboOutput">
```

```
        <!-- Text output will be displayed here -->
    </div>
</div>

<h2>Diagram:</h2>
<svg id="cboDiagram" width="700" height="600"></svg>

<script>
    // --- Helper SVG Functions --- (Include drawBlock, drawTenBlock, createText from previous examples) ---
    // Simplified drawBlock for this viz
    function drawBlock(svg, x, y, size, fill, className = 'block') {
        const rect = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
        rect.setAttribute('x', x); rect.setAttribute('y', y);
        rect.setAttribute('width', size); rect.setAttribute('height', size);
        rect.setAttribute('fill', fill);
        rect.setAttribute('class', className);
        svg.appendChild(rect);
        return { x, y, width: size, height: size, type: 'o', cx: x + size/2, cy: y + size/2 }; // Add center point
    }

     function drawTenBlock(svg, x, y, width, height, fill, unitBlockSize) { // Keep vertical ten block
        const group = document.createElementNS("http://www.w3.org/2000/svg", 'g');
        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('class', 'ten-block-bg␣block');
        group.appendChild(backgroundRect);

        for (let i = 0; i < 10; i++) {
            const unitBlock = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
            unitBlock.setAttribute('x', x); unitBlock.setAttribute('y', y + i * unitBlockSize);
            unitBlock.setAttribute('width', unitBlockSize); unitBlock.setAttribute('height', unitBlockSize);
            unitBlock.setAttribute('fill', fill);
            unitBlock.setAttribute('class', 'unit-block-inner');
            group.appendChild(unitBlock);
        }
        svg.appendChild(group);
         return { x, y, width, height, type: 't', cx: x + width/2, cy: y + height/2};
    }

     function createText(svg, x, y, textContent, className = 'diagram-label', anchor = 'start') {
        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', anchor);
        text.textContent = textContent;
        svg.appendChild(text);
    }

     function createCurvedArrow(svg, x1, y1, x2, y2, cx, cy, arrowClass='redistribute-
         arrow', headClass='redistribute-arrow-head', arrowSize=4) {
        const path = document.createElementNS("http://www.w3.org/2000/svg", 'path');
        path.setAttribute('d', `M ${x1} ${y1} Q ${cx} ${cy} ${x2} ${y2}`);
        path.setAttribute('class', arrowClass);
        svg.appendChild(path);

        const arrowHead = document.createElementNS("http://www.w3.org/2000/svg", 'path');
        const dx = x2 - cx; const dy = y2 - cy;
        const angleRad = Math.atan2(dy, dx);
        const angleDeg = angleRad * (180 / Math.PI);
        arrowHead.setAttribute('d', `M 0 0 L ${arrowSize} ${arrowSize/2} L ${arrowSize} $
            {-arrowSize/2} Z`);
        arrowHead.setAttribute('class', headClass);
        arrowHead.setAttribute('transform', `translate(${x2}, ${y2}) rotate(${angleDeg +
            180})`);
        svg.appendChild(arrowHead);
    }
    // --- End Helper Functions ---

    // --- Main CBO Automaton Function ---
    document.addEventListener('DOMContentLoaded', function() {
        const outputElement = document.getElementById('cboOutput');
        const groupsInput = document.getElementById('cboGroups');
        const itemsInput = document.getElementById('cboItems');
        const diagramSVG = document.getElementById('cboDiagram');

        if (!outputElement || !groupsInput || !itemsInput || !diagramSVG) {
            console.error("Required␣HTML␣elements␣not␣found!");
            return;
        }

        // Function to convert number to word (simple version)
        function numberToWord(num) {
            const words = ["Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
                "Eight", "Nine", "Ten", "Eleven", "Twelve"];
            if (num >= 0 && num < words.length) {
                return words[num];
            }
            return num.toString(); // Fallback to numeral if > 12
        }


        window.runCBOAutomaton = function() {
            try {
                const numGroups = parseInt(groupsInput.value);
                const itemsPerGroup = parseInt(itemsInput.value);

```

```
145             if (isNaN(numGroups) || isNaN(itemsPerGroup) || numGroups <= 0 ||
                    itemsPerGroup <= 0) {
146                 outputElement.textContent = "Please␣enter␣valid␣positive␣numbers";
147                 diagramSVG.innerHTML = ''; return;
148             }
149
150             const totalItems = numGroups * itemsPerGroup;
151             const finalTensCount = Math.floor(totalItems / 10);
152             const finalOnesCount = totalItems % 10;
153             const numGroupsWord = numberToWord(numGroups); // Get word for groups
154
155             // --- Generate Text Notation (Matching PDF) ---
156             let output = `<h2>Conversion to Bases and Ones (CBO) - Notation</h2>\n\n`;
157             output += `<p class="notation-line␣problem">${numGroupsWord}  ${
                    itemsPerGroup} = ?</p>\n`;
158
159             if (itemsPerGroup < 10 && numGroups > 1) {
160                 const neededPerGroup = 10 - itemsPerGroup;
161                 const groupsToComplete = numGroups - 1; // Try to complete all but one
                        group
162                 const totalNeeded = groupsToComplete * neededPerGroup;
163                 // Find how many ones are left in the last group after donating
164                 const onesLeftInLastGroup = itemsPerGroup - totalNeeded;
165
166                 if (onesLeftInLastGroup >= 0) { // Check if the last group had enough
167                     output += `<p class="notation-line">= ${numberToWord(
                            groupsToComplete)}  ${itemsPerGroup} + ${itemsPerGroup}</p>\n`;
168                     output += `<p class="notation-line">= ${numberToWord(
                            groupsToComplete)}  ${itemsPerGroup} + ${totalNeeded} + ${
                            onesLeftInLastGroup}</p>\n`; // Show split of last group
169                     output += `<p class="notation-line">= ${numberToWord(
                            groupsToComplete)}  (${itemsPerGroup} + ${neededPerGroup}) + ${
                            onesLeftInLastGroup}</p>\n`; // Show distribution
170                     output += `<p class="notation-line">= ${numberToWord(
                            groupsToComplete)}  10 + ${onesLeftInLastGroup}</p>\n`;
171                     output += `<p class="notation-line">= ${groupsToComplete * 10} + $
                            {onesLeftInLastGroup}</p>\n`;
172                     output += `<p class="notation-line">= ${totalItems}</p>\n`;
173                 } else {
174                     // Logic for needing more than one group to decompose is more
                            complex
175                     // For simplicity, just show the direct calculation result for text
                            if simple decomp fails
176                     output += `<p class="notation-line">= ${totalItems} (Direct
                            Calculation)</p>\n`;
177                 }
178             } else {
179                 // If itemsPerGroup >= 10 or only one group, direct calculation is
                        simpler notation
180                 output += `<p class="notation-line">= ${totalItems} (Direct
                        Calculation)</p>\n`;
181             }
182
183
```

```
            outputElement.innerHTML = output;
            typesetMath();

            // --- Draw Diagram ---
            drawCBODiagram('cboDiagram', numGroups, itemsPerGroup, finalTensCount,
                finalOnesCount);

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

    function drawCBODiagram(svgId, numGroups, itemsPerGroup, finalTensCount,
        finalOnesCount) {
        const svg = document.getElementById(svgId);
        if (!svg) return;
        svg.innerHTML = '';

        const svgWidth = parseFloat(svg.getAttribute('width'));
        const svgHeight = parseFloat(svg.getAttribute('height'));
        const blockUnitSize = 10;
        const tenBlockWidth = blockUnitSize;
        const tenBlockHeight = blockUnitSize * 10;
        const blockSpacing = 4;
        const groupSpacingX = 30; // Increase spacing between initial groups
        const sectionSpacingY = 150; // Increased vertical space
        const startX = 30;
        let currentY = 40;
        const colorGroup = 'teal';
        const colorResultTen = 'lightgreen';
        const colorResultOne = 'gold';
        const arrowOffsetY = -15; // Y offset for arrow start/end above blocks
        const arrowControlOffsetY = -60; // How high the arrow arc goes

        // --- 1. Initial Groups Visualization ---
        createText(svg, startX, currentY, `Initial State: ${numberToWord(numGroups)}
            groups of ${itemsPerGroup}`);
        currentY += 30;
        let currentX = startX;
        let section1MaxY = currentY;
        let initialGroupsData = []; // Store positions of initial blocks [{group: g,
            item: i, x, y, size}]

        for (let g = 0; g < numGroups; g++) {
            let groupStartX = currentX;
            let itemYOffset = 0;
            // Draw items vertically within the group
            for (let i = 0; i < itemsPerGroup; i++) {
                let blockInfo = drawBlock(svg, currentX, currentY + itemYOffset,
                    blockUnitSize, blockUnitSize, colorGroup);
                initialGroupsData.push({ group: g, item: i, x: blockInfo.x, y:
                    blockInfo.y, size: blockUnitSize, cx: blockInfo.cx, cy: blockInfo.
                    cy });
```

```
231                 itemYOffset += blockUnitSize + blockSpacing;
232             }
233             currentX = groupStartX + blockUnitSize + groupSpacingX; // Next group
                    starts after one block width + spacing
234             section1MaxY = Math.max(section1MaxY, currentY + itemYOffset);
235         }
236
237         // --- 2. Redistribution Arrows (Conceptual) ---
238         // Only draw if redistribution is feasible (S<10, N>1, and last group has
                enough)
239         const neededPerGroup = (itemsPerGroup < 10) ? 10 - itemsPerGroup : 0;
240         const groupsToComplete = numGroups - 1;
241         const totalNeeded = groupsToComplete * neededPerGroup;
242         const onesLeftInLastGroup = itemsPerGroup - totalNeeded;
243
244         if (neededPerGroup > 0 && onesLeftInLastGroup >= 0 && numGroups > 1) {
245             // Find blocks in the last group to be the source
246             let sourceBlocks = initialGroupsData.filter(d => d.group === numGroups -
                    1).slice(0, totalNeeded); // Get the first 'totalNeeded' blocks from
                    the last group
247             let targetGroups = initialGroupsData.filter(d => d.group < numGroups - 1)
                    ;
248
249             let sourceIndex = 0;
250             for (let g = 0; g < groupsToComplete; g++) {
251                 // Find the top-most block of the target group 'g'
252                 let targetBlock = targetGroups.find(d => d.group === g && d.item ===
                        itemsPerGroup -1); // Top item in the target group
253                 if (targetBlock && sourceIndex < sourceBlocks.length) {
254                     let sourceBlock = sourceBlocks[sourceIndex];
255                     // Draw arrow from source block to above target block
256                     createCurvedArrow(svg,
257                         sourceBlock.cx, sourceBlock.cy, // Start center of source
                                block
258                         targetBlock.cx, targetBlock.y + arrowOffsetY, // End
                                slightly above target block
259                         (sourceBlock.cx + targetBlock.cx) / 2, sourceBlock.cy +
                                arrowControlOffsetY // Control point for arc
260                     );
261                     sourceIndex++;
262                 }
263                 // We need to distribute 'neededPerGroup' to each target group
264                 // This loop just draws one arrow per target group for simplicity
265                 // A more complex viz could draw neededPerGroup arrows per target
                        group
266             }
267         }
268
269
270         currentY = section1MaxY + sectionSpacingY;
271
272
273         // --- 3. Final Result Visualization (Base-10 Blocks) ---
274         let finalSum = numGroups * itemsPerGroup; // Recalculate for safety
```

```
            createText(svg, startX, currentY, `Final Result (Converted to Base-10): ${finalSum}`);
            currentY += 30;
            currentX = startX;
            let section2MaxY = currentY;

            for (let i = 0; i < finalTensCount; i++) { drawTenBlock(svg, currentX, currentY, tenBlockWidth, tenBlockHeight, colorResultTen, blockUnitSize); currentX += tenBlockWidth + blockSpacing; section2MaxY = Math.max(section2MaxY, currentY + tenBlockHeight); }
            // Align final ones vertically
            let finalOnesY = currentY + Math.max(0, tenBlockHeight - (finalOnesCount * (blockUnitSize + blockSpacing))); // Align bottom or top? Align top here.
            for (let i = 0; i < finalOnesCount; i++) { drawBlock(svg, currentX, finalOnesY + i * (blockUnitSize + blockSpacing), blockUnitSize, blockUnitSize, colorResultOne); section2MaxY = Math.max(section2MaxY, finalOnesY + (i+1)*(blockUnitSize+blockSpacing)); }
            currentX += blockUnitSize + blockSpacing; // Add spacing after ones

        } // End drawCBODiagram


        function typesetMath() { /* Placeholder */ }

        // Initialize on page load
        runCBOAutomaton();

    }); // End DOMContentLoaded
</script>

</body>
</html>
```

References

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