Counting in Base 10

Theodore M. Savich

March 30, 2025

1 Diagonalizing the Count

1.1 Sublation in Counting: From Tallies to Base Systems

Counting is not merely an accumulation of marks – it is a process that both *preserves* and *transforms* prior determinations. In Hegelian terms, this movement is called *sublation* (Aufhebung), the simultaneous *negation*, *preservation*, and *uplift* of what came before. In mathematical practice, sublation is most clearly seen in the way base systems reorganize quantities into new structural units.

Consider a simple act of tally counting. If one were to count to nine using tally marks, the representation would appear as:

$$|||||||||$$

Each tally stands independently as a discrete marker of a counted object that mirrors the "world of ones" reflected in von Neumann ordinals. They could just go on and on, accumulating indefinitely. While it is more normal to represent a transformation at 5 units, let us instead live in base ten. When ten is reached, the representation undergoes an important transformation:

$$\cancel{|||||||||}$$

The previous nine marks are not erased. They are not 'gone.' But they are *negated* and *uplifted* into a new structural form. Out of the many ones, there is now one ten. This is a mathematical instance of sublation. The prior elements are not discarded. They are reorganized in a higher-level composition. The transition from loose tallies to a single "ten" does not merely introduce a new symbol; it alters how the prior marks are understood. They are still 'present,' but they no longer function as isolated entities.

So, using base systems involves "two views" of a number - but under the hood is very basic version of a diagonalizing function, δ, that lets an element reference the whole system it's part of. Ten loose ones is a "many", one 10 is a "one". Diagonalization is, therefore, a way of thinking about the problem of the one and the many.

2 Understanding the Recursive Nature of Counting

Counting in base 10 involves incrementing digits and managing composition across multiple place values:

- **Units (Ones):** $10^0 = 1$
- **Tens:** $10^1 = 10$

- **Hundreds:** $10^2 = 100$
- **Thousands:** $10^3 = 1,000$, etc.

The recursive process for counting follows these steps:

1. Increment the units digit.
2. If the units digit reaches 10, reset it to 0 and increment the tens digit.
3. Repeat this process recursively for higher place values as needed.

This recursive nature allows for counting indefinitely by reusing the same increment and composition logic for each digit.

3 Why Use a Pushdown Automaton (PDA)?

A Pushdown Automaton (PDA) is suitable for modeling recursive counting due to its ability to use a stack for memory. Here's why:

- **Finite State Automaton (FSA):** Lacks the memory to handle arbitrary-length counts and composition.
- **Pushdown Automaton (PDA):** Uses a stack to provide additional memory, enabling nested operations like composition in counting.
- **Turing Machine:** While capable, it is more complex than needed for this task.

A PDA's stack can represent digit states and manage composition recursively, making it an appropriate choice.

4 Designing the Pushdown Automaton for Recursive Counting

4.1 Components of the PDA

The PDA is defined by the following components:

- **States:**
    1. q_{start}: Start state.
    2. q_{count}: Handles incrementing and composition.
    3. q_{output}: Outputs the current count.
    4. q_{accept}: Accepting state (optional for finite counting).
- **Input Alphabet:** $\Sigma = \{\emptyset\}$ (each $\emptyset$ represents a unit to count).
- **Stack Alphabet:** $\Gamma = \{\#, D_0, D_1, D_2, \ldots\}$:
    - $\#$ is the bottom-of-stack marker.
    - D_n represents the digit at the n^{th} place (e.g., D_0 for units, D_1 for tens).

4.2 Automaton Behavior

The PDA operates with the following behavior:

1. **Initialization:**
   - Start in q_{start}, push # onto the stack as a marker.
   - Push D_0 onto the stack to represent the initial digit (0).
   - Transition to q_{count} to begin counting.
2. **Counting and Handling composition:**
   - In q_{count}, increment the current digit D_n.
   - If $D_n < 9$, replace it with D_{n+1} to represent the incremented value.
   - If $D_n = 9$, reset it to D_0 and handle the composition by incrementing or pushing D_{n+1} onto the stack.
3. **Output the Current Count:**
   - In q_{output}, traverse the stack to read the current count from top to bottom.
   - Transition back to q_{count} for the next input.

5 Conceptual State Diagram

The diagram below illustrates the key states and transitions for recursive counting using a PDA.

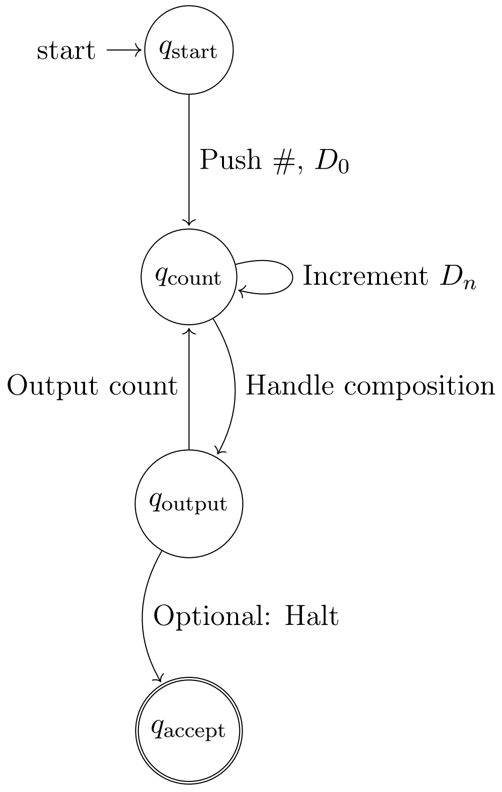


Figure 1: Conceptual State Diagram for Recursive Counting with a PDA

6 Detailed Example Execution: Counting from 0 to 12

This section demonstrates how the PDA counts from 0 to 12.

1. **Start:** Stack = $\#D_0$ (represents 0)
2. **Input 1 ($\emptyset$):**
   - Increment D_0 to D_1.
   - Stack = $\#D_1$ (represents 1)
3. **Input 2 ($\emptyset$):**
   - Increment D_1 to D_2.
   - Stack = $\#D_2$ (represents 2)
4. **... Continue up to Input 9 ($\emptyset$):**
   - Increment D_8 to D_9.
   - Stack = $\#D_9$ (represents 9)
5. **Input 10 ($\emptyset$):**
   - D_0 resets to D_0, representing composition.
   - Push D_1 onto the stack to increment the tens place.
   - Stack = $\#D_0D_1$ (represents 10)
6. **Input 11 ($\emptyset$):**
   - Increment D_0 to D_1.
   - Stack = $\#D_1D_1$ (represents 11)
7. **Input 12 ($\emptyset$):**
   - Increment D_0 to D_2.
   - Stack = $\#D_2D_1$ (represents 12)

7 Recursive Handling of composition

To manage composition:

1. If a digit reaches 9, reset it to 0 and increment the next higher digit.
2. If no higher digit exists, push a new digit onto the stack to represent a new place value.
3. Repeat this process recursively as needed for higher digits.

8 Formal Transition Function

The transition function δ for the PDA is defined as follows:

- $\delta(q_{\text{start}}, \epsilon, \epsilon) = (q_{\text{count}}, \#D_0)$
- $\delta(q_{\text{count}}, \emptyset, D_n) = \begin{cases} (q_{\text{output}}, D_{n+1}), & \text{if } n < 9 \\ (q_{\text{count}}, D_0), & \text{if } n = 9 \text{ (reset and composition)} \end{cases}$
- $\delta(q_{\text{output}}, \epsilon, \gamma) = (q_{\text{count}}, \gamma)$

9 Modeling Recursion in the PDA

The stack in a PDA enables recursion by storing the current state of each digit:

- The top of the stack represents the least significant digit.
- As digits are incremented, composition operations recursively modify higher digits.

9.1 Recursive Handling Mechanism

When a digit is incremented and reaches 9:

1. The PDA resets it to 0.
2. The composition is handled by incrementing or adding the next higher digit.
3. If all digits require resetting (e.g., from 999 to 1000), a new digit is pushed onto the stack.

10 Example: Counting from 999 to 1000

1. **Initial Stack:** $\#D_9D_9D_9$ (represents 999)
2. **Input ($\emptyset$):**
   - Reset D_0 to D_0.
   - Increment D_1, resulting in D_0D_0.
   - Increment D_2 similarly until pushing a new digit D_1.

11 Modeling the Recursive Aspect in Detail

The PDA uses the stack to simulate recursion by representing each digit's state in a Last-In-First-Out (LIFO) order. This section provides a deeper explanation of how the PDA can manage an unbounded number of digits using recursive composition and stack operations.

11.1 Recursive Counting Mechanism

1. **Digit Incrementation:**

- The PDA increments the current top digit on the stack, which represents the units place.
- If the digit is less than 9, it simply replaces the current stack symbol with the incremented value (e.g., $D_n \to D_{n+1}$).
- If the digit equals 9, it resets the digit to D_0 and triggers a composition to the next higher place value.

2. **Handling composition:**

- When a composition occurs, the PDA checks if there is a higher digit already on the stack.
- If there is an existing higher digit, the PDA increments it.
- If no higher digit exists (only the bottom marker # is present), the PDA pushes a new digit D_1 onto the stack, representing the tens place.
- This process continues recursively, allowing the PDA to manage arbitrarily large numbers by dynamically expanding the stack.

11.2 State Reusability and Recursive Simulation

The PDA uses a finite set of states (q_{count}, q_{output}, etc.) repeatedly for each digit operation:

- **State Reusability:** The same states handle different digit positions due to the stack's dynamic nature.
- **Recursive Simulation:** The stack's LIFO behavior enables the PDA to handle the composition and increment operations recursively without needing new states for each digit position.
- **Infinite Counting Capability:** Despite having a finite number of states, the PDA can count infinitely by pushing more digits onto the stack as needed.

12 Formal Transition Function for Recursive Counting

The transition function δ encapsulates the recursive logic required for counting in base 10. Here is the detailed definition:

1. **Initialization:**

$$\delta(q_{\text{start}}, \epsilon, \epsilon) = (q_{\text{count}}, \#D_0)$$

This pushes the bottom marker # and the initial digit D_0 onto the stack.

2. **Counting State (q_{count}):**

$$\delta(q_{\text{count}}, \emptyset, D_n) = \begin{cases} (q_{\text{output}}, D_{n+1}), & \text{if } n < 9 \\ (q_{\text{count}}, D_0), & \text{if } n = 9 \text{ (reset and composition)} \end{cases}$$

This transition increments the top digit or resets it and initiates a composition if needed.

3. **composition Handling (q_{compose}):**

- **If there is a higher digit on the stack:** Increment it.

$$\delta(q_{\text{compose}}, \epsilon, D_m) = (q_{\text{output}}, D_{m+1})$$

- **If no higher digit exists (only # is present):** Push a new digit D_1 onto the stack.

$$\delta(q_{\text{compose}}, \epsilon, \#) = (q_{\text{output}}, D_1\#)$$

4. **Output State (q_{output}):**

$$\delta(q_{\text{output}}, \epsilon, \gamma) = (q_{\text{count}}, \gamma)$$

This transition reads the current stack configuration to output the count and returns to the counting state.

13 Illustrative Example: Counting from 999 to 1000

The PDA handles multi-digit composition using the stack to simulate recursive behavior. Here is a step-by-step illustration of counting from 999 to 1000.

1. **Initial Stack Configuration:** $\#D_9D_9D_9$ (represents 999)
2. **Input 1 ($\emptyset$):**
   - Increment D_0 from 9 to 0 (reset).
   - Trigger composition to the next digit.
3. **composition to Next Digit:**
   - Increment D_1 from 9 to 0 (reset).
   - Continue the composition process to D_2.
4. **composition to D_2:**
   - Increment D_2 from 9 to 0 (reset).
   - Since no higher digit exists, push a new digit D_1 onto the stack.
5. **Final Stack Configuration:** $\#D_0D_0D_0D_1$ (represents 1000)

The PDA successfully manages the transition from 999 to 1000 by recursively applying composition operations through the stack.

14 Recursive Handling for Arbitrarily Large Numbers

The PDA is designed to handle an unbounded number of digits. Here's how it achieves this:
1. **Stack as Dynamic Memory:**

- The stack grows to accommodate additional digits as the number increases.
- Each digit is pushed onto the stack, simulating a new recursive level.

2. **composition Across Multiple Digits:**

- composition is handled recursively from the least significant to the most significant digit.
- If all digits are at their maximum value (e.g., $999\dots9$), new stack symbols are pushed to extend the number (e.g., $1000\dots0$).

3. **Infinite Loop for Counting:**

- The PDA can loop between q_{count} and q_{output} indefinitely, enabling it to count an infinite sequence of inputs.
- The recursive nature of the stack allows the PDA to manage numbers of any size without requiring additional states.

15 Practical Considerations and Limitations

While the theoretical PDA can handle an infinite counting sequence, practical implementations have limitations:

- **Stack Limitations:** In real-world applications, memory constraints may limit the size of the stack.
- **Abstract Output Mechanism:** The output of the PDA is conceptualized as reading the stack's state; in practice, a mechanism would be needed to convert the stack symbols into readable numerical output.
- **Finite Resources:** Although the PDA theoretically counts infinitely, actual implementations are restricted by finite computational resources.

16 Conclusion

This Pushdown Automaton model demonstrates how recursion and stack memory can be used to simulate counting in base 10. By leveraging the stack for dynamic digit handling and composition management, the PDA can handle numbers of arbitrary length using a finite set of states.

16.1 Key Takeaways

- **Recursion via Stack Operations:** The stack's LIFO structure enables recursive operations for digit incrementation and composition.
- **Finite State Reusability:** A finite number of states can support infinite counting when combined with stack memory.
- **Theoretical and Practical Implications:** Understanding the PDA model provides insights into computational theory and number systems, while practical limitations highlight the challenges of implementing infinite processes.

HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Counting with Two Representations</title>
    <style>
        body { font-family: sans-serif; line-height: 1.6; }
        .representation-section { margin-bottom: 20px; padding-bottom: 10px; border-bottom
            : 1px solid #eee; }
        .box { /* Style for individual box */
            display: inline-block;
            width: 18px; height: 18px; margin: 1px;
            background-color: lightblue; border: 1px solid #666;
            vertical-align: middle;
        }
        .rectangle-10 { /* Style for composed ten rectangle */
            display: inline-block;
            width: 198px; height: 18px; margin: 1px;
            background-color: lightgreen; border: 1px solid #333;
            vertical-align: middle;
            text-align: center; line-height: 18px;
            font-size: 12px; font-weight: bold;
        }
        .clickable { cursor: pointer; } /* Indicate clickable */
        .clickable:hover { border-color: red; outline: 1px solid red; /* Add outline on
            hover */ }

        .tally-svg-group { /* Style for the SVG container */
            display: inline-block; /* Allow spacing */
            vertical-align: middle;
            margin-right: 5px; /* Space between tally groups */
            height: 30px; /* Set height for alignment */
        }
        .tally-svg-group line { /* Style for lines within SVG */
            stroke: black;
            stroke-width: 2;
        }

        .button-row { margin: 10px 0; }
        button { padding: 5px 10px; font-size: 1em; margin-right: 5px; }
        #numericValue { font-size: 1.5em; font-weight: bold; color: darkblue; }

    </style>
</head>
<body>

    <h1>Counting in Base 10 with Two Representations</h1>
    <p>Illustrating sublation: how 10 individual 'ones' become 1 'ten'.</p>

    <div class="button-row">
        <button onclick="decrementCount()" id="decrementBtn">- Decrement</button>
        <button onclick="incrementCount()">+ Increment</button>
    </div>

    <p><strong>Numerical Value:</strong> <span id="numericValue">0</span></p>

    <div class="representation-section">
        <strong>Boxes Representation:</strong> (Click on '10' representations to toggle)<
            br />
        <span id="boxesDisplay"></span>
    </div>

    <div class="representation-section">
        <strong>Tally Representation:</strong> (Click on '10' representations to toggle)<
            br />
        <span id="tallyDisplay"></span>
    </div>


    <script>
        let count = 0;
        let tenAsSingleBox = false;
        let tenAsSlashTally = false; // Use this state for the diagonal slash tally

        const numericValueSpan = document.getElementById("numericValue");
        const boxesContainer = document.getElementById("boxesDisplay");
        const tallyContainer = document.getElementById("tallyDisplay");
        const decrementBtn = document.getElementById("decrementBtn");

        function incrementCount() { count++; updateDisplay(); }
        function decrementCount() { if (count > 0) { count--; updateDisplay(); } }

        function toggleTenBoxRepresentation() {
            if (count === 10) { tenAsSingleBox = !tenAsSingleBox; updateDisplay(); }
        }
        function toggleTenTallyRepresentation() {
             if (count === 10) { tenAsSlashTally = !tenAsSlashTally; updateDisplay(); } //
                  Toggle new state
        }

        // --- SVG Tally Group Drawing Function ---
        function drawTallyGroupSVG(parentContainer, isSlashed = true, isClickable = false)
             {
            const svgNS = "http://www.w3.org/2000/svg";
            const svg = document.createElementNS(svgNS, "svg");
            const verticalBarHeight = 25;
            const verticalBarSpacing = 4;
            const groupWidth = (verticalBarSpacing + 2) * 9 + 2; // 9 bars + spacing +
                stroke width
            const svgWidth = groupWidth + (isSlashed ? 10 : 0); // Extra width for slash
                overhang? Adjust as needed
             const svgHeight = 30;

            svg.setAttribute("width", svgWidth);
            svg.setAttribute("height", svgHeight);
            svg.setAttribute("class", "tally-svg-group" + (isClickable ? "␣clickable" : ""));
            if (isClickable) {
                svg.onclick = toggleTenTallyRepresentation;
                svg.setAttribute("title", isSlashed ? "1␣Ten␣(Composed␣-␣Click␣to␣decompose)" : "10␣Ones␣(Click␣to␣compose)");
            } else {
                 svg.setAttribute("title", isSlashed ? "1␣Ten␣(Composed)" : "10␣Ones");
            }


            // Draw 10 vertical bars if NOT slashed
            if (!isSlashed) {
                for (let i = 0; i < 10; i++) {
                    const line = document.createElementNS(svgNS, "line");
                    const x = i * (verticalBarSpacing + 2) + 1; // +1 for stroke width offset
                    line.setAttribute("x1", x); line.setAttribute("y1", (svgHeight - verticalBarHeight) / 2);
                    line.setAttribute("x2", x); line.setAttribute("y2", (svgHeight + verticalBarHeight) / 2);
                    svg.appendChild(line);
                }
            } else { // Draw 9 vertical + 1 diagonal slash
                 for (let i = 0; i < 9; i++) {
                    const line = document.createElementNS(svgNS, "line");
                    const x = i * (verticalBarSpacing + 2) + 1;
                    line.setAttribute("x1", x); line.setAttribute("y1", (svgHeight - verticalBarHeight) / 2);
                    line.setAttribute("x2", x); line.setAttribute("y2", (svgHeight + verticalBarHeight) / 2);
                    svg.appendChild(line);
                }
                // Draw diagonal slash
                const slash = document.createElementNS(svgNS, "line");
                const startX = 0; // Start slightly before first bar
                const startY = (svgHeight + verticalBarHeight) / 2 + 2; // Start lower left
                const endX = groupWidth + 4; // End slightly after last bar
                const endY = (svgHeight - verticalBarHeight) / 2 - 2; // End upper right
                slash.setAttribute("x1", startX); slash.setAttribute("y1", startY);
                slash.setAttribute("x2", endX); slash.setAttribute("y2", endY);
                svg.appendChild(slash);
            }

            parentContainer.appendChild(svg);
        }
        // --- End SVG Tally Group ---

        function updateDisplay() {
            numericValueSpan.textContent = count;
            decrementBtn.disabled = (count === 0);

            // --- Update Boxes ---
            boxesContainer.innerHTML = ""; // Clear previous
            const boxTens = Math.floor(count / 10);
            const boxOnes = count % 10;

            for (let t = 0; t < boxTens; t++) {
                const isToggleable = (count === 10 && t === 0); // Only clickable at EXACTLY 10
                if (isToggleable && tenAsSingleBox) {
                    const rect = document.createElement("div");
                    rect.className = "rectangle-10 clickable";
                    rect.title = "1 Ten (Click to decompose)";
                    rect.onclick = toggleTenBoxRepresentation;
                    boxesContainer.appendChild(rect);
                } else if (isToggleable && !tenAsSingleBox) {
                    for (let i = 0; i < 10; i++) {
                        const box = document.createElement("div");
                        box.className = "box clickable";
                        box.title = "1 One (Click to compose)";
                        box.onclick = toggleTenBoxRepresentation;
                         boxesContainer.appendChild(box);
                    }
                 } else { // For tens groups when count > 10 or default state at 10
                     if (tenAsSingleBox) { // Use the *current* toggle state for display
                        const rect = document.createElement("div");
                        rect.className = "rectangle-10";
                        rect.title = "1 Ten";
                        boxesContainer.appendChild(rect);
                    } else {
                        for (let i = 0; i < 10; i++) {
                            const box = document.createElement("div");
                            box.className = "box";
                            box.title = "1 One";
                            boxesContainer.appendChild(box);
                        }
                    }
                 }
                 // Add spacer between tens groups or before ones
                  if (boxTens > 0 && boxOnes > 0 || t < boxTens - 1) {
                     const spacer = document.createElement("span");
                     spacer.style.display = "inline-block"; spacer.style.width = "8px";
                     boxesContainer.appendChild(spacer);
                 }
            }
            // Draw ones boxes
            for (let i = 0; i < boxOnes; i++) {
                const box = document.createElement("div");
                box.className = "box";
                boxesContainer.appendChild(box);
            }


            // --- Update Tallies ---
            tallyContainer.innerHTML = ""; // Clear previous
            const tallyTens = Math.floor(count / 10);
            const tallyOnes = count % 10;

            // Draw tens groups using SVG
            for (let t = 0; t < tallyTens; t++) {
                const isToggleable = (count === 10 && t === 0); // Clickable only at count
                     10
                const useSlashed = isToggleable ? tenAsSlashTally : tenAsSlashTally; //
                    Draw based on toggle state
                drawTallyGroupSVG(tallyContainer, useSlashed, isToggleable);

                 // No extra spacer needed, margin on SVG handles it
            }

            // Draw remainder (ones) tallies as simple text |
            if (tallyOnes > 0) {
                const onesSpan = document.createElement("span");
                onesSpan.className = "tally-mark";
                onesSpan.textContent = "|".repeat(tallyOnes);
                tallyContainer.appendChild(onesSpan);
            }

        } // End of updateDisplay

        // Initialize the display on page load
        updateDisplay();

    </script>

</body>
</html>
```

Addition Strategies: Chunking by Bases and Ones

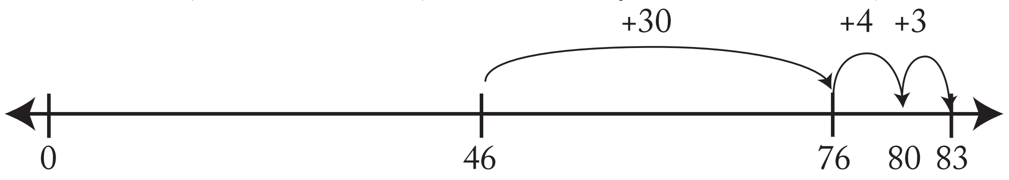
Compiled by: Theodore M. Savich

April 1, 2025

Transcript

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

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



Notation Representing Sarah's Solution:

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

Description of Strategy:

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

Description of Strategy

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

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

Automaton Type

Finite State Automaton (FSA) with basic arithmetic capability.

Formal Description of the Automaton

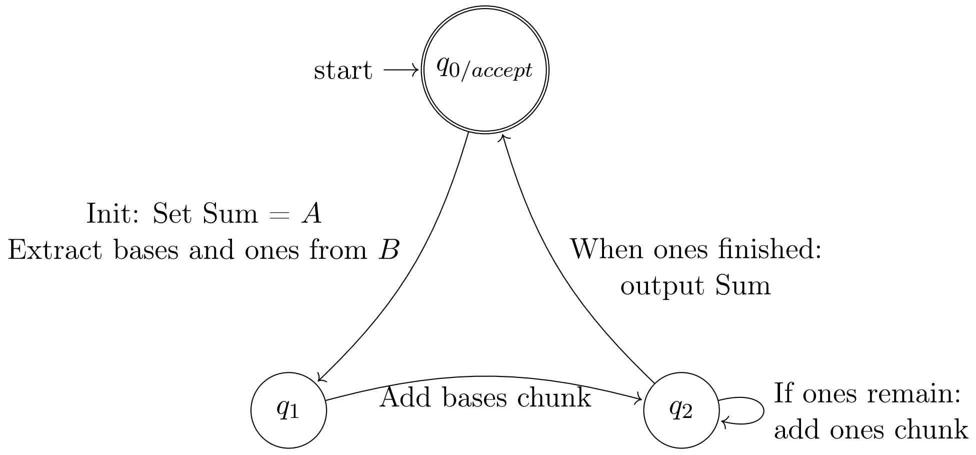
We define the automaton as the tuple

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

where:

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

Automaton Diagram for Chunking by Bases and Ones



HTML Implementation

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

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

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

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

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

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

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

    if (!outputElement || !diagramChunkingSVG) {
        console.warn('Element␣chunkingOutput␣or␣diagramChunkingSVG␣not␣found');
        return;
    }

    window.runChunkingAutomaton = function() {
        try {
            const addend1 = parseInt(chunkingAddend1Input.value);
            const addend2 = parseInt(chunkingAddend2Input.value);
            if (isNaN(addend1) || isNaN(addend2)) {
                outputElement.textContent = 'Please␣enter␣valid␣numbers␣for␣both␣addends';
                return;
            }

            let output = `<h2>Chunking by Bases and Ones (Flexible)</h2>\n\n`;
            output += `<p><strong>Problem:</strong> ${addend1} + ${addend2}</p>\n\n`;

            let tensToAddTotal = Math.floor(addend2 / 10) * 10;
            let onesToAddTotal = addend2 % 10;

            output += `Step 1: Split ${addend2} into ${tensToAddTotal} (tens) + ${
                onesToAddTotal} (ones)\n\n`;

            let currentSum = addend1;
            const chunkSteps = [];
            let stepCounter = 2;

            // --- Strategy Decision: Add Ones First or Tens First? ---
            const addOnesFirstDecision = Math.random() < 0.3; // 30% chance to add ones
                first (if possible)
            let onesAddedFirst = false;

            if (addOnesFirstDecision && onesToAddTotal > 0) {
                // Try adding ones first to make the next ten
                const onesToNextTenInitial = (10 - (currentSum % 10)) % 10;
                if (onesToNextTenInitial > 0 && onesToAddTotal >= onesToNextTenInitial) {
                     output += `Step ${stepCounter}: Add ones chunk first to make a ten\n`;
                     stepCounter++;
                     chunkSteps.push({
                        from: currentSum,
                        to: currentSum + onesToNextTenInitial,
                        label: `+${onesToNextTenInitial}`
                    });
                    output += `<p>${currentSum} + ${onesToNextTenInitial} = ${currentSum +
                        onesToNextTenInitial} (Making the next ten)</p>\n`;
                    currentSum += onesToNextTenInitial;
                    onesToAddTotal -= onesToNextTenInitial;
                    onesAddedFirst = true; // Flag that we adjusted ones already
                    output += '\n';
                }
            }

            // --- Tens Chunking (Potentially after adding some ones) ---
```

```
if (tensToAddTotal > 0) {
    output += `Step ${stepCounter}: Add tens chunk(s)\n`;
    stepCounter++;

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

        let tensChunk = 0;

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

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

// --- Remaining Ones Chunking (If not added first or some left over) ---
```

```
        if (onesToAddTotal > 0) {
             output += `Step ${stepCounter}: Add remaining ones chunk(s)\n`;

            // Strategic ones (make next ten) - might happen again if tens landed
                awkwardly
            const onesToNextTen = (10 - (currentSum % 10)) % 10;

            if (onesToNextTen > 0 && onesToAddTotal >= onesToNextTen) {
                // Chunk 1: Reach the next ten
                chunkSteps.push({
                    from: currentSum,
                    to: currentSum + onesToNextTen,
                    label: `+${onesToNextTen}`
                });
                output += `<p>${currentSum} + ${onesToNextTen} = ${currentSum +
                    onesToNextTen} (Making the next ten)</p>\n`;
                currentSum += onesToNextTen;
                onesToAddTotal -= onesToNextTen;

                // Chunk 2: Add the rest
                if (onesToAddTotal > 0) {
                     chunkSteps.push({
                        from: currentSum,
                        to: currentSum + onesToAddTotal,
                        label: `+${onesToAddTotal}`
                    });
                    output += `<p>${currentSum} + ${onesToAddTotal} = ${currentSum +
                        onesToAddTotal}</p>\n`;
                    currentSum += onesToAddTotal;
                    onesToAddTotal = 0;
                }
            } else if (onesToAddTotal > 0) {
                // Add all remaining ones
                chunkSteps.push({
                    from: currentSum,
                    to: currentSum + onesToAddTotal,
                    label: `+${onesToAddTotal}`
                });
                output += `<p>${currentSum} + ${onesToAddTotal} = ${currentSum +
                    onesToAddTotal}</p>\n`;
                currentSum += onesToAddTotal;
                onesToAddTotal = 0;
            }
             output += '\n';
        }


        output += `Result: ${addend1} + ${addend2} = ${currentSum}`;
        outputElement.innerHTML = output;
        typesetMath();

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

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

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

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

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

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

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

        // Determine the actual min and max values shown *after* the break
        let minValAfterBreak = addend1;
        let maxValAfterBreak = finalSum;
        chunkSteps.forEach(step => {
            minValAfterBreak = Math.min(minValAfterBreak, step.from, step.to);
            maxValAfterBreak = Math.max(maxValAfterBreak, step.from, step.to);
        });


        if (addend1 > scaleBreakThreshold) {
            displayRangeStart = minValAfterBreak - 10; // Start range slightly before min value shown after break
            scaleStartX = startX + 30; // Leave space for break symbol
            drawScaleBreak = true;
            drawScaleBreakSymbol(svg, scaleStartX - 15, numberLineY); // Draw break symbol
        } else {
            displayRangeStart = 0; // Start from 0 if no break
        }

        const displayRangeEnd = maxValAfterBreak + 10; // End range slightly after max value shown
        const displayRange = Math.max(displayRangeEnd - displayRangeStart, 1); // Avoid division by zero if range is 0
        const scale = (endX - scaleStartX) / displayRange;

        // Function to convert value to X coordinate based on scale
        function valueToX(value) {
             if (value < displayRangeStart && drawScaleBreak) {
                 // Values before the effective start are compressed near the break symbol
                 return scaleStartX - 10; // Place them just before the break starts visually
             }
              // Ensure values stay within the visible range after the break starts
             const scaledValue = scaleStartX + (value - displayRangeStart) * scale;
             return Math.min(scaledValue, endX); // Cap at endX
        }

        // Draw Ticks and Labels for relevant points
        function drawTickAndLabel(value, index) {
            const x = valueToX(value);
             if (x < scaleStartX - 5 && value !== 0) return; // Don't draw ticks in compressed area unless it's 0 or very close to break

            const tick = document.createElementNS('http://www.w3.org/2000/svg', 'line');
            tick.setAttribute('x1', x);
            tick.setAttribute('y1', numberLineY - tickHeight / 2);
            tick.setAttribute('x2', x);
            tick.setAttribute('y2', numberLineY + tickHeight / 2);
            tick.setAttribute('class', 'number-line-tick');
            svg.appendChild(tick);
            const labelOffset = labelOffsetBase * (index % 2 === 0 ? 1 : -1.5);
            createText(svg, x, numberLineY + labelOffset, value.toString(), 'number-line-label');
        }

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

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

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

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

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

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

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


        // --- Helper SVG drawing functions --- (Keep these the same) ---
         function createText(svg, x, y, textContent, className) {
            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'); // Keep middle align for labels
            text.setAttribute('font-size', '12px');
            text.textContent = textContent;
            svg.appendChild(text);
        }
```

```

         function drawScaleBreakSymbol(svg, x, y) {
            const breakOffset = 4; // How far apart the lines are
            const breakHeight = 8; // How tall the zig-zag is
            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); // Swapped x1/x2
            breakLine2.setAttribute('y1', y - breakHeight);
            breakLine2.setAttribute('x2', x - breakOffset); // Swapped x1/x2
            breakLine2.setAttribute('y2', y + breakHeight);
            breakLine2.setAttribute('class', 'number-line-break');
            svg.appendChild(breakLine2);
        }

        function createJumpArrow(svg, x1, y1, x2, y2, jumpArcHeight) {
            const path = document.createElementNS('http://www.w3.org/2000/svg', 'path');
            const cx = (x1 + x2) / 2;
            const cy = y1 - jumpArcHeight; // Arc is above the line
            path.setAttribute('d', `M ${x1} ${y1} Q ${cx} ${cy} ${x2} ${y1}`);
            path.setAttribute('class', 'jump-arrow');
            svg.appendChild(path);

            // Arrowhead
            const jumpArrowHead = document.createElementNS('http://www.w3.org/2000/svg', 'path');
            const dx = x2 - cx; // Approx direction vector
            const dy = y1 - cy;
            const angleRad = Math.atan2(dy, dx);
            const angleDeg = angleRad * (180 / Math.PI);
            jumpArrowHead.setAttribute('class', 'jump-arrow-head');
            jumpArrowHead.setAttribute('d', `M 0 0 L ${arrowSize} ${arrowSize/2} L ${arrowSize} ${-arrowSize/2} Z`);
            jumpArrowHead.setAttribute('transform', `translate(${x2}, ${y1}) rotate(${angleDeg + 180})`);
            svg.appendChild(jumpArrowHead);
        }

        function drawStoppingPoint(svg, x, y, labelText, labelOffsetBase = 20, index = 0) {
            const circle = document.createElementNS('http://www.w3.org/2000/svg', 'circle');
            circle.setAttribute('cx', x);
            circle.setAttribute('cy', y);
            circle.setAttribute('r', 4);
            circle.setAttribute('class', 'stopping-point');
            svg.appendChild(circle);

            // Use the provided y parameter instead of numberLineY
            if (labelText) {
                // Add staggering based on index to prevent overlap with large values
                const labelOffset = labelOffsetBase * (index % 2 === 0 ? 1.5 : -1.8);
                createText(svg, x, y + labelOffset, labelText, 'number-line-label');
            }
        }
    }

    function typesetMath() {
        // Placeholder
    }

});
</script>

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

<script>
    function openPdfViewer() {
        // Opens the PDF documentation for the strategy.
        window.open('../SAR_ADD_CHUNKING.pdf', '_blank');
    }
</script>
</html>
```

References

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

Addition Strategies: Rearranging to Make Bases (RMB)

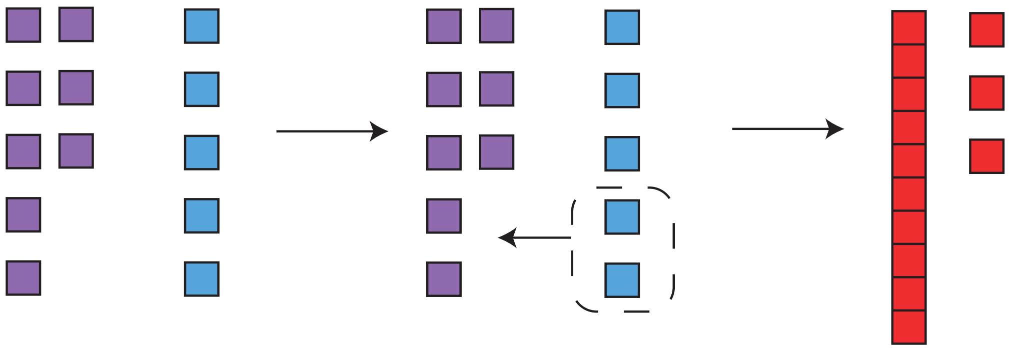
Compiled by: Theodore M. Savich

March 31, 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

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

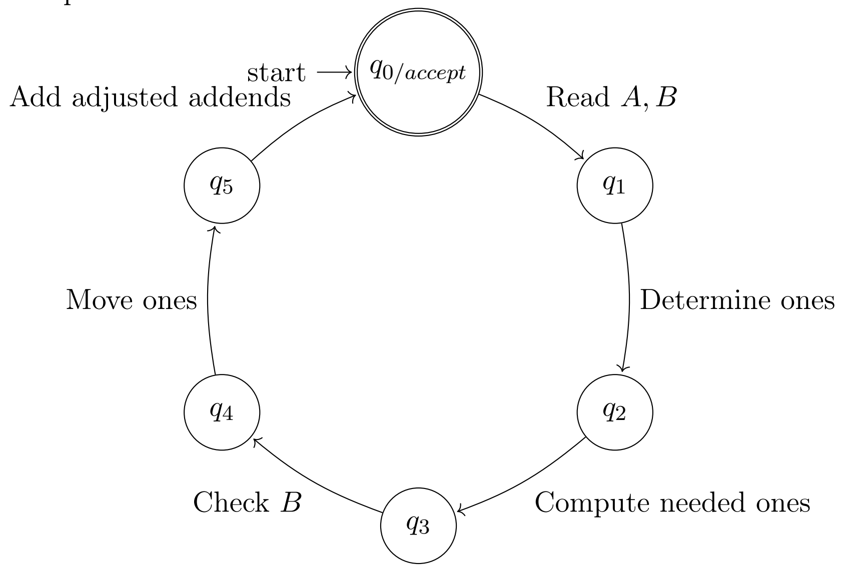
is defined by the following key transitions:

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 -->

    <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;

            if (toMakeBase === 0) {
                output += `${addend1} is already a multiple of 10.<br>`;
                output += 'Directly␣add:␣${addend1}␣+␣${addend2}␣=␣${addend1␣+␣addend2}';
                rmbOutputElement.textContent = output;
                drawRMBDiagram('diagramRMBSVG', addend1, addend2, toMakeBase, addend1,
                    addend2, addend1 + addend2);
                return;
            }

            if (addend2 < toMakeBase) {
                output += 'Cannot␣make␣a␣base␣from␣${addend1}␣because␣${addend2}␣is␣too␣
                    small␣to␣provide␣the␣needed␣${toMakeBase}␣units.<br>';
                output += 'Directly␣add:␣${addend1}␣+␣${addend2}␣=␣${addend1␣+␣addend2}';
                rmbOutputElement.textContent = output;
                drawRMBDiagram('diagramRMBSVG', addend1, addend2, toMakeBase, addend1,
                    addend2, addend1 + addend2);
                return;
            }

            // Apply RMB strategy
            const newAddend1 = addend1 + toMakeBase;
            const newAddend2 = addend2 - toMakeBase;
            const result = newAddend1 + newAddend2;

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

            rmbOutputElement.innerHTML = output;

            // Draw RMB Diagram
            drawRMBDiagram('diagramRMBSVG', addend1, addend2, toMakeBase, newAddend1,
                newAddend2, result);


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


    function drawRMBDiagram(svgId, addend1, addend2, toMakeBase, newAddend1, newAddend2,
        result) {
        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 colors = ['lightblue', 'lightcoral']; // Colors for addend blocks

```

```
        // --- 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, 'purple');
            addend1X += tenBlockWidth + blockSpacing;
        }
        let a1_onesX = addend1X;
        for (let i = 0; i < a1_ones; i++) {
            drawBlock(svg, a1_onesX, currentY + i*(blockUnitSize + blockSpacing), blockUnitSize, blockUnitSize, 'purple');
        }
        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, 'blue');
            addend2X += tenBlockWidth + blockSpacing;
        }
        const addend2OnesX = addend2X;
        let movedBlockTopY = null, movedBlockBottomY = null;
        for (let i = 0; i < a2_ones; i++) {
            drawBlock(svg, addend2OnesX, currentY + i*(blockUnitSize + blockSpacing), blockUnitSize, blockUnitSize, 'blue');
            if (i < toMakeBase) {
                if (movedBlockTopY === null) {
                    movedBlockTopY = currentY + i*(blockUnitSize + blockSpacing);
                }
                movedBlockBottomY = currentY + i*(blockUnitSize + blockSpacing) + blockUnitSize;
            }
        }
        currentY += tenBlockHeight + sectionSpacingY; // Move down for the rearranged addends section

        // --- Rearranged Addends ---
        createText(svg, startX+20, currentY, `Rearranged to Make Base: ${newAddend1} + ${newAddend2}`); // Label
        currentY += 30; // Space after label

        // Draw Rearranged Addend 1 Blocks (Tens only, since newAddend1 is a multiple of 10)
```

```javascript
        let currentX_newAddend1 = startX;
        const newAddend1_tens = Math.floor(newAddend1 / 10);
        for (let i = 0; i < newAddend1_tens; i++) {
             drawTenBlock(svg, currentX_newAddend1, currentY, tenBlockWidth, tenBlockHeight, 'red');
             currentX_newAddend1 += tenBlockWidth + blockSpacing;
        }
        // Draw Rearranged Addend 2 Blocks (Split into tens and ones)
        const newAddend2_tens = Math.floor(newAddend2 / 10);
        const newAddend2_ones = newAddend2 % 10;
        let currentX_newAddend2 = currentX_newAddend1 + 40; // Horizontal spacing after newAddend1 blocks
        for (let i = 0; i < newAddend2_tens; i++) {
            drawTenBlock(svg, currentX_newAddend2, currentY, tenBlockWidth, tenBlockHeight, 'blue');
            currentX_newAddend2 += tenBlockWidth + blockSpacing;
        }
        for (let i = 0; i < newAddend2_ones; i++) {
            drawBlock(svg, currentX_newAddend2, currentY + i*(blockUnitSize + blockSpacing), blockUnitSize, blockUnitSize, 'blue');
        }

        // --- Curved Arrow ---
        if (toMakeBase > 0 && addend2 >= toMakeBase && movedBlockTopY !== null) {
            // Arrow from center of moved (vertical) ones in addend2 to the rearranged tens block assembly
            const arrowStartX = addend2OnesX + blockUnitSize/2;
            const arrowStartY = movedBlockTopY + (movedBlockBottomY - movedBlockTopY) / 2;
            const arrowEndX = startX + tenBlockWidth/2;
            const arrowEndY = currentY; // top of rearranged addend1 blocks
            // Use control point midway vertically between arrowStartY and arrowEndY
            const controlY = (arrowStartY + arrowEndY) / 2;
            createCurvedArrow(svg, arrowStartX, arrowStartY, arrowEndX, arrowEndY, arrowEndX, controlY);
            createText(svg, arrowEndX + 30, controlY + 35, `${toMakeBase} moved`);
        }

        // --- 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);
        }

    }

});
    </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].

Subtraction Strategies: Counting On/Back By Bases and then Ones (CBBO)

Compiled by: Theodore M. Savich

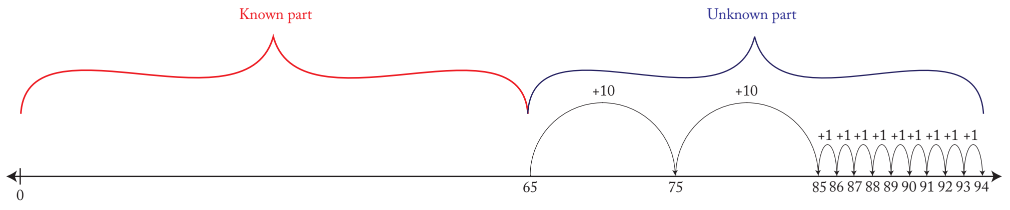
March 28, 2025

Transcript

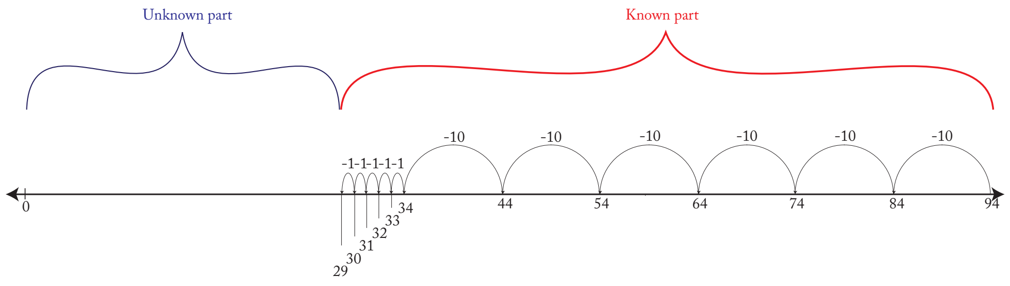
Video from Carpenter et al. (1999). Strategy descriptions and examples adapted from Hackenberg (2025).

- **Teacher:** Earl had a collection of 65 bird feathers, on a trip to a marsh he found lots more feathers to put in his collection. Now he has 94 feathers in his collection. How many feathers did Earl find at the marsh?
- **Rita** So he had what?
- **Teacher:** He started off with, 65 feathers.
- **Rita:** 1,2,3,4,5,6 1,2,3,4,5. And then he had how many?
- **Teacher:** Well, he had 65 bird feathers. On a trip to a marsh, he found lots more and he put them in his collection. Now he has 94.
- **Rita:** Well, I can 65, 75, 85. How many did he find?
- **Teacher:** Well, that's my question for you. How many did he find? He ends up with 94.
- **Rita:** And 85,86,87,88,89,90, 91,92,93,94 and so the answer is 20, 21, 22, 23, 24, 25, 26, 27, 28, 29.
- **Teacher** Nice work.

Rita's Way: Counting On by Bases and then Ones (COBO)



Alternatively, Rita could have Counted Back by Bases and Ones (CBBO)



Notation Representing Rita's Solution:

$$\begin{aligned} 65 + (10) &= 75 \\ 75 + (10) &= 85 \\ 85 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 &= 94 \\ 10 + 10 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 &= 29 \end{aligned}$$

Description of Strategy:

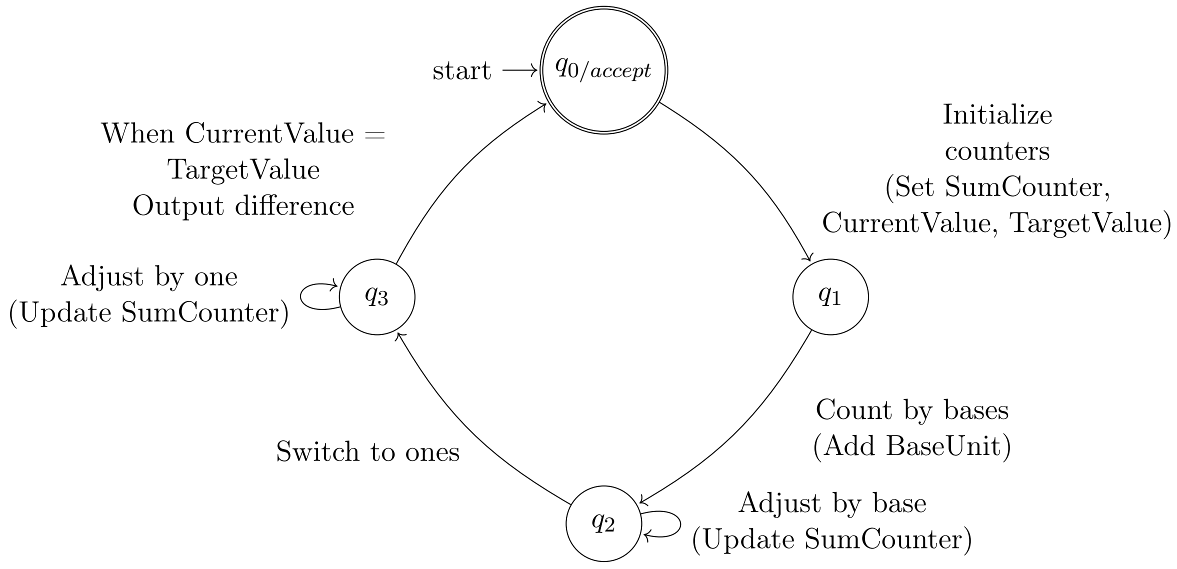
Objective: Description of Counting On by Bases and Then Ones (COBO) Begin with one of the numbers. Break the other number into its base units and its ones. Then, "count on" by adding each base unit one at a time, followed by each individual one.

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.

Automaton Diagram for Counting On or Back by Bases and Then Ones


start
$q_{0/accept}$
Initialize counters (Set SumCounter, CurrentValue, TargetValue)
q_1
Count by bases (Add BaseUnit)
q_2
Adjust by base (Update SumCounter)
Switch to ones
q_3
Adjust by one (Update SumCounter)
When CurrentValue = TargetValue Output difference

HTML Implementation

```html
<!DOCTYPE html>
<html>
<head>
    <title>Subtraction Strategies: Counting Back By Bases and Ones (CBBO)</title>
    <style>
        body { font-family: sans-serif; }
        #diagramCBBOSVG { border: 1px solid #d3d3d3; }
        #outputContainer { margin-top: 20px; }
        .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; stroke: purple; stroke-width: 1.5; } /* CBBO color */
        .jump-arrow-head { fill: purple; stroke: purple; } /* CBBO color */
        .jump-label { font-size: 10px; text-anchor: middle; fill: purple; } /* CBBO color
            */
        .tens-jump-label { font-size: 12px; text-anchor: middle; fill: purple; }
        .stopping-point { fill: red; stroke: black; stroke-width: 1; }
        .number-line-arrow { fill: black; stroke: black; }
        .extended-tick { stroke: black; stroke-width: 1; } /* Reuse COBO style */
    </style>
</head>
<body>

<h1>Subtraction Strategies: Counting Back By Bases and Then Ones (CBBO)</h1>

<div>
    <label for="cbboMinuend">Minuend:</label>
    <input type="number" id="cbboMinuend" value="94"> <!-- Example from PDF -->
</div>
<div>
    <label for="cbboSubtrahend">Subtrahend:</label>
    <input type="number" id="cbboSubtrahend" value="29"> <!-- 94 - 65 = 29 -->
</div>

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

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

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

<script>
document.addEventListener('DOMContentLoaded', function() {
    const outputElement = document.getElementById('cbboOutput');
    const cbboMinuendInput = document.getElementById('cbboMinuend');
    const cbboSubtrahendInput = document.getElementById('cbboSubtrahend');
    const diagramCBBOSVG = document.getElementById('diagramCBBOSVG');
```

```

// --- Helper Functions (Keep createText, drawTick, drawScaleBreakSymbol,
    createJumpArrow, drawStoppingPoint from previous corrected versions) ---
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'); // Center labels
    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',
     arrowSize = 5) { // Removed default color, use CSS
     const path = document.createElementNS('http://www.w3.org/2000/svg', 'path');
     const cx = (x1 + x2) / 2;
     const cy = y1 - jumpArcHeight; // Arc is above the line
     path.setAttribute('d', `M ${x1} ${y1} Q ${cx} ${cy} ${x2} ${y1}`); // Use y1 for
         end point to land on line
     path.setAttribute('class', `jump-arrow`); // Rely on CSS for color
     svg.appendChild(path);

     // Arrowhead
     const arrowHead = document.createElementNS('http://www.w3.org/2000/svg', 'path');
     const dx = x2 - cx;
     const dy = y1 - cy; // Use y1 for angle calculation
     const angleRad = Math.atan2(dy, dx);
     let angleDeg = angleRad * (180 / Math.PI);
     arrowHead.setAttribute('class', `jump-arrow-head`); // Rely on CSS for color

     if (direction === 'forward') {
        angleDeg += 180; // Point right
        arrowHead.setAttribute('d', `M 0 0 L ${arrowSize} ${arrowSize/2} L ${arrowSize} ${-arrowSize/2} Z`);
        arrowHead.setAttribute('transform', `translate(${x2}, ${y1}) rotate(${angleDeg})`);
     } else { // backward
        // angleDeg already points left-ish from Q curve end
        arrowHead.setAttribute('d', `M 0 0 L ${-arrowSize} ${arrowSize/2} L ${-arrowSize} ${-arrowSize/2} Z`); // Pointy part is at (0,0)
         // We want to rotate to align with the curve's end direction
        arrowHead.setAttribute('transform', `translate(${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');
 }
// --- End Helper Functions ---

// --- Main CBBO Automaton Function ---
window.runCBBOAutomaton = function() {
    try {
       const minuend = parseInt(cbboMinuendInput.value);
       const subtrahend = parseInt(cbboSubtrahendInput.value); // Amount to subtract
       if (isNaN(minuend) || isNaN(subtrahend)) {
           outputElement.textContent = 'Please enter valid numbers for Minuend and Subtrahend';
           diagramCBBOSVG.innerHTML = '';
           return;
       }
        if (subtrahend > minuend) {
           outputElement.textContent = 'Subtrahend cannot be greater than Minuend for CBBO.';
           diagramCBBOSVG.innerHTML = '';
           return;
       }

       let output = `<h2>Counting Back by Bases and Ones (CBBO)</h2>\n\n`;
       output += `<p><strong>Problem:</strong> ${minuend} - ${subtrahend}</p>\n\n`;
```

```

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

            output += `Step 1: Split subtrahend ${subtrahend} into ${tensToSubtract} + ${onesToSubtract}\n\n`;

            let currentVal = minuend;
            const tensSteps = [];
            if (tensToSubtract > 0) {
                output += 'Step␣2:␣Count␣back␣by␣tens\n';
                for (let i = 10; i <= tensToSubtract; i += 10) {
                    tensSteps.push({ from: currentVal, to: currentVal - 10, action: 'Subtract␣10' });
                    currentVal -= 10;
                }
                tensSteps.forEach(step => {
                    output += `<p>${step.from} - 10 = ${step.to}</p>\n`; // Simplified text
                });
                output += '\n';
            }

            const onesSteps = [];
            if (onesToSubtract > 0) {
                output += `Step ${tensToSubtract > 0 ? '3' : '2'}: Count back by ones\n`;
                for (let i = 1; i <= onesToSubtract; i++) {
                    onesSteps.push({ from: currentVal, to: currentVal - 1, action: 'Subtract␣1' });
                    currentVal -= 1;
                }
                onesSteps.forEach(step => {
                    output += `<p>${step.from} - 1 = ${step.to}</p>\n`; // Simplified text
                });
                output += '\n';
            }

            const finalDifference = currentVal; // The final landing spot IS the difference
            output += `Result: ${minuend} - ${subtrahend} = ${finalDifference}`;
            outputElement.innerHTML = output;
            typesetMath();

            // Draw the diagram
            drawCBBONumberLineDiagram(diagramCBBOSVG, minuend, subtrahend, tensSteps, onesSteps, finalDifference);


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

```

```
function drawCBBONumberLineDiagram(svg, minuend, subtrahend, tensSteps, onesSteps,
    finalDifference) {
    if (!svg || typeof svg.setAttribute !== 'function') { 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; // Center vertically
    const tickHeight = 10;
    const labelOffsetBase = 20;
    const jumpHeight = 30; // Consistent jump height for CBBO
    const jumpLabelOffset = 15;
    const arrowSize = 5;
    const scaleBreakThreshold = 40;

    // Determine range for scaling
    let diagramMin = finalDifference;
    let diagramMax = minuend;

    // Calculate scale and handle potential break (near 0, before diagramMin)
    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);
           const labelOffset = labelOffsetBase * (index % 2 === 0 ? 1 : -1.5); // Stagger
           createText(svg, x, numberLineY + labelOffset, value.toString(), 'number-line-
               label');
       }

       // Collect all points to draw ticks for
       let allPoints = new Set([minuend, finalDifference]); // Start and end
       tensSteps.forEach(step => allPoints.add(step.to));
       onesSteps.forEach(step => allPoints.add(step.to));
       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 tens jumps (Backward)
       tensSteps.forEach((step, index) => {
           const x1 = valueToX(step.from);
           const x2 = valueToX(step.to);
           if (x1 <= scaleStartX || x2 < scaleStartX) return; // Skip if outside visible
               range

            const staggerOffset = index % 2 === 0 ? 0 : jumpHeight * 0.5;
            createJumpArrow(svg, x1, numberLineY, x2, numberLineY, jumpHeight +
                staggerOffset, 'backward', arrowSize);
            createText(svg, (x1 + x2) / 2, numberLineY - (jumpHeight + staggerOffset) -
                jumpLabelOffset, '-10', 'tens-jump-label');
        });

        // Draw ones jumps (Backward)
        onesSteps.forEach((step, index) => {
            const x1 = valueToX(step.from);
            const x2 = valueToX(step.to);
             if (x1 <= scaleStartX || x2 < scaleStartX) return; // Skip if outside visible
                 range

            const staggerOffset = (tensSteps.length + index) % 2 === 0 ? 0 : jumpHeight *
                0.5; // Continue staggering
            createJumpArrow(svg, x1, numberLineY, x2, numberLineY, jumpHeight +
                staggerOffset, 'backward', arrowSize);
            createText(svg, (x1 + x2) / 2, numberLineY - (jumpHeight + staggerOffset) -
                jumpLabelOffset, '-1', 'jump-label');
        });

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

    function typesetMath() { /* Placeholder */ }

    // Initial run on page load
    runCBBOAutomaton();

});
</script>

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

<script>
    function openPdfViewer() {
        // Opens the PDF documentation for the strategy.
        window.open('../PDF_Documentation_Of_Strategies/SAR_SUB_COBO.pdf', '_blank');
    }
</script></html>
</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].

Subtraction Strategies: Decomposition

Compiled by: Theodore M. Savich

March 30, 2025

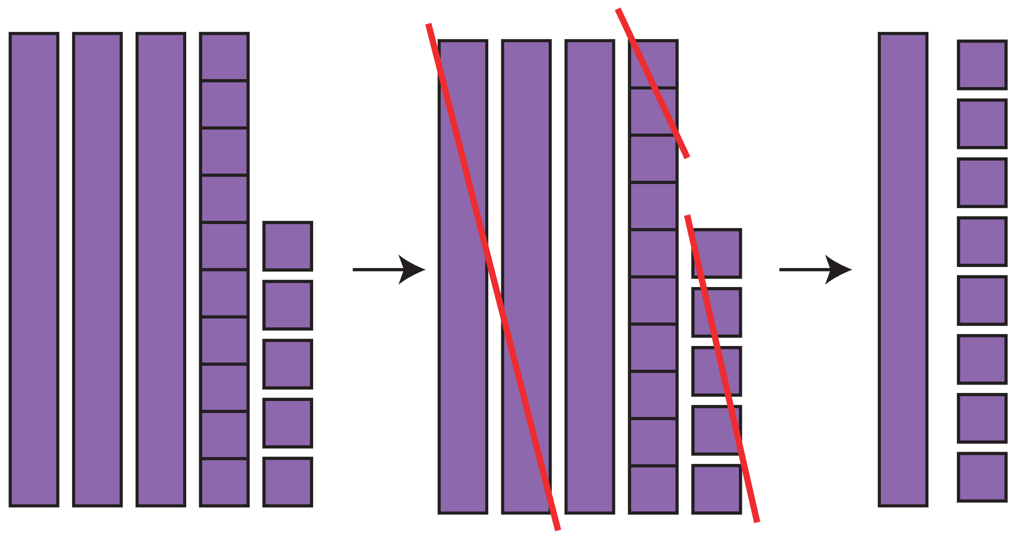
Transcript

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

- **Teacher:** Lucy ordered 45 cupcakes for her birthday. At the party, her guests ate 27 cupcakes, how many cupcakes did she have left? [BACKGROUND]
- **Joel:** This is 10, this is 10, this is 10, this is 10 and this is five.18.
- **Teacher:** Explain to us what you did there.
- **Joel:** I have, this is 10, this is 10, this is 10, and this is five. So I take away 20 and I take away five. I take away two more. So they enter and then I counted these and those, and so the answer was 18.
- **Teacher:** Nice work

Notation Representing Joel's Solution:

$$\begin{aligned}
&47-27\\
&45-20=25\\
&\ 25-7=?\\
&\qquad 2\text{ tens }+5\text{ ones }-7\text{ ones}\\
&1\text{ ten }+1\text{ ten }+5\text{ ones }-7\text{ ones}\\
&\qquad \downarrow \text{DECOMPOSE}\\
&1\text{ ten }+10\text{ ones }+5\text{ ones }-7\text{ ones}\\
&\qquad 1\text{ ten }+8\text{ ones }+\underbrace{7\text{ ones }-7\text{ ones}}_{=0}\\
&\qquad 1\text{ ten }+8\text{ ones}
\end{aligned}$$



Notation Representing Joel's Solution: Imagine representing both numbers by their base units and ones. Begin by subtracting the base components, then subtract the ones. If there aren't enough ones available in the larger number to subtract the ones from the smaller number (while keeping the result positive), break one base unit into its individual ones. Finally, remove only the exact number of ones required to complete the subtraction.

Decomposition

Description of Strategy

- **Objective:** Decompose a base unit from the minuend into ones to have enough ones to subtract the ones in the subtrahend.

Automaton Type

Pushdown Automaton (PDA): Needed to handle the decomposition process and keep track of 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:

- $Q = \{q_{0/accept},\, q_1,\, q_2,\, q_3,\, q_4,\, q_5\}$ is the set of states.
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is the input alphabet.
- $\Gamma = \{Z_0\} \cup \{b \mid b \in \mathbb{N}\}$ is the stack alphabet, where Z_0 is the initial stack symbol and b represents a base unit (e.g., 10 in base-ten).
- $q_{0/accept}$ is the start state, which is also the accept state.
- $F = \{q_{0/accept}\}$ is the set of accepting states.

The transition function δ is defined as:

1. $\delta(q_{0/accept}, \text{“}M, S\text{”}, Z_0) = \{(q_1, Z_0)\}$
   (Split the minuend M and subtrahend S into their base and ones components.)

2. $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$
   (Compare the ones in M and S.)

3. $\delta(q_2, \varepsilon, Z_0) = \{(q_3, b\,Z_0)\}$
   (If the ones in M are insufficient, decompose a base unit b into ones.)

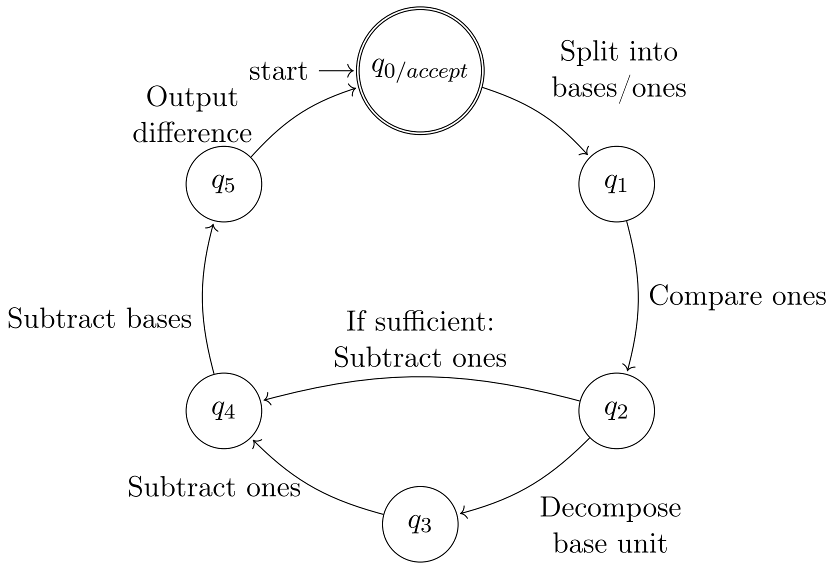
4. $\delta(q_2, \varepsilon, Z_0) = \{(q_4, Z_0)\}$
   (If the ones in M are sufficient, proceed to subtract ones.)

5. $\delta(q_3, \varepsilon, b) = \{(q_4, b)\}$
   (After decomposition, subtract the ones.)

6. $\delta(q_4, \varepsilon, Z_0) = \{(q_5, Z_0)\}$
   (Subtract the bases.)

7. $\delta(q_5, \varepsilon, Z_0) = \{(q_{0/accept}, Z_0)\}$
   (Output the final difference.)

Automaton Diagram for Decomposition



HTML Implementation

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="" xml:lang="">
<head>
  <meta charset="utf-8" />
  <meta name="generator" content="pandoc" />
  <meta name="viewport" content="width=device-width,␣initial-scale=1.0,␣user-scalable=yes
    " />
  <meta name="author" content="Theodore␣M.␣Savich" />
  <title>Subtraction Strategies: Decomposition</title>
  <style>
    html {
      color: #1a1a1a;
      background-color: #fdfdfd;
    }
    body {
      margin: 0 auto;
      max-width: 36em;
      padding-left: 50px;
      padding-right: 50px;
      padding-top: 50px;
      padding-bottom: 50px;
      hyphens: auto;
      overflow-wrap: break-word;
      text-rendering: optimizeLegibility;
      font-kerning: normal;
    }
    @media (max-width: 600px) {
      body {
        font-size: 0.9em;
        padding: 12px;
      }
      h1 {
        font-size: 1.8em;
      }
    }
    @media print {
      html {
        background-color: white;
      }
      body {
        background-color: transparent;
        color: black;
        font-size: 12pt;
      }
      p, h2, h3 {
        orphans: 3;
        widows: 3;
      }
      h2, h3, h4 {
        page-break-after: avoid;
      }
    }
```

```css
p {
  margin: 1em 0;
}
a {
  color: #1a1a1a;
}
a:visited {
  color: #1a1a1a;
}
img {
  max-width: 100%;
}
svg {
  height: auto;
  max-width: 100%;
}
h1, h2, h3, h4, h5, h6 {
  margin-top: 1.4em;
}
h5, h6 {
  font-size: 1em;
  font-style: italic;
}
h6 {
  font-weight: normal;
}
ol, ul {
  padding-left: 1.7em;
  margin-top: 1em;
}
li > ol, li > ul {
  margin-top: 0;
}
blockquote {
  margin: 1em 0 1em 1.7em;
  padding-left: 1em;
  border-left: 2px solid #e6e6e6;
  color: #606060;
}
code {
  font-family: Menlo, Monaco, Consolas, 'Lucida␣Console', monospace;
  font-size: 85%;
  margin: 0;
  hyphens: manual;
}
pre {
  margin: 1em 0;
  overflow: auto;
}
pre code {
  padding: 0;
  overflow: visible;
  overflow-wrap: normal;
}
```

```
.sourceCode {
 background-color: transparent;
 overflow: visible;
}
hr {
  background-color: #1a1a1a;
  border: none;
  height: 1px;
  margin: 1em 0;
}
table {
  margin: 1em 0;
  border-collapse: collapse;
  width: 100%;
  overflow-x: auto;
  display: block;
  font-variant-numeric: lining-nums tabular-nums;
}
table caption {
  margin-bottom: 0.75em;
}
tbody {
  margin-top: 0.5em;
  border-top: 1px solid #1a1a1a;
  border-bottom: 1px solid #1a1a1a;
}
th {
  border-top: 1px solid #1a1a1a;
  padding: 0.25em 0.5em 0.25em 0.5em;
}
td {
  padding: 0.125em 0.5em 0.25em 0.5em;
}
header {
  margin-bottom: 4em;
  text-align: center;
}
#TOC li {
  list-style: none;
}
#TOC ul {
  padding-left: 1.3em;
}
#TOC > ul {
  padding-left: 0;
}
#TOC a:not(:hover) {
  text-decoration: none;
}
code{white-space: pre-wrap;}
span.smallcaps{font-variant: small-caps;}
div.columns{display: flex; gap: min(4vw, 1.5em);}
div.column{flex: auto; overflow-x: auto;}
div.hanging-indent{margin-left: 1.5em; text-indent: -1.5em;}
```

```
    /* The extra [class] is a hack that increases specificity enough to
       override a similar rule in reveal.js */
    ul.task-list[class]{list-style: none;}
    ul.task-list li input[type="checkbox"] {
      font-size: inherit;
      width: 0.8em;
      margin: 0 0.8em 0.2em -1.6em;
      vertical-align: middle;
    }
  </style>
  <script
  src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-chtml-full.js"
  type="text/javascript"></script>
</head>
<body>
<header id="title-block-header">
<h1 class="title">Subtraction Strategies: Decomposition</h1>
<p class="author">Theodore M. Savich</p>
</header>
<h2 class="unnumbered" id="decomposition">Decomposition</h2>
<h3 class="unnumbered" id="description-of-strategy">Description of
Strategy</h3>
<ul>
<li><p><strong>Objective:</strong> Decompose a base unit from the
minuend into ones to have enough ones to subtract the ones in the
subtrahend.</p></li>
</ul>
<h3 class="unnumbered" id="automaton-type">Automaton Type</h3>
<p><strong>Pushdown Automaton (PDA)</strong>: Needed to handle the
borrowing (decomposition) process and keep track of base units.</p>
<h3 class="unnumbered" id="formal-description-of-the-automaton">Formal
Description of the Automaton</h3>
<p>We define the PDA as the 7-tuple <span class="math␣display">\[M =
(Q,\,\Sigma,\,\Gamma,\,\delta,\,q_{0/accept},\,Z_0,\,F)\]</span>
where:</p>
<ul>
<li><p><span class="math␣inline">\(Q = \{q_{0/accept},\, q_1,\, q_2,\,
q_3,\, q_4,\, q_5\}\)</span> is the set of states.</p></li>
<li><p><span class="math␣inline">\(\Sigma =
\{0,1,2,3,4,5,6,7,8,9\}\)</span> is the input alphabet.</p></li>
<li><p><span class="math␣inline">\(\Gamma = \{Z_0\} \cup \{b \mid b \in
\mathbb{N}\}\)</span> is the stack alphabet, where <span
class="math␣inline">\(Z_0\)</span> is the initial stack symbol and <span
class="math␣inline">\(b\)</span> represents a base unit (e.g., 10 in
base-ten).</p></li>
<li><p><span class="math␣inline">\(q_{0/accept}\)</span> is the start
state, which is also the accept state.</p></li>
<li><p><span class="math␣inline">\(F = \{q_{0/accept}\}\)</span> is the
set of accepting states.</p></li>
</ul>
<p>The transition function <span class="math␣inline">\(\delta\)</span>
is defined as:</p>
<ol>
<li><p><span class="math␣inline">\(\delta(q_{0/accept},\,
\text{‘‘}M,S\text{&#39;&#39;},\, Z_0) = \{(q_1,\, Z_0)\}\)</span><br />
(Split the minuend <span class="math␣inline">\(M\)</span> and subtrahend
<span class="math␣inline">\(S\)</span> into their base and ones
components.)</p></li>
<li><p><span class="math␣inline">\(\delta(q_1,\, \varepsilon,\, Z_0) =
\{(q_2,\, Z_0)\}\)</span><br />
(Compare the ones in <span class="math␣inline">\(M\)</span> and <span
class="math␣inline">\(S\)</span>.)</p></li>
<li><p><span class="math␣inline">\(\delta(q_2,\, \varepsilon,\, Z_0) =
\{(q_3,\, b\,Z_0)\}\)</span><br />
(If the ones in <span class="math␣inline">\(M\)</span> are insufficient,
decompose a base unit <span class="math␣inline">\(b\)</span> into
ones.)</p></li>
<li><p><span class="math␣inline">\(\delta(q_2,\, \varepsilon,\, Z_0) =
\{(q_4,\, Z_0)\}\)</span><br />
(If the ones in <span class="math␣inline">\(M\)</span> are sufficient,
proceed to subtract ones.)</p></li>
<li><p><span class="math␣inline">\(\delta(q_3,\, \varepsilon,\, b) =
\{(q_4,\, b)\}\)</span><br />
(After decomposition, subtract the ones.)</p></li>
<li><p><span class="math␣inline">\(\delta(q_4,\, \varepsilon,\, Z_0) =
\{(q_5,\, Z_0)\}\)</span><br />
(Subtract the bases.)</p></li>
<li><p><span class="math␣inline">\(\delta(q_5,\, \varepsilon,\, Z_0) =
\{(q_{0/accept},\, Z_0)\}\)</span><br />
(Output the final difference.)</p></li>
</ol>
<h3 class="unnumbered"
id="automaton-diagram-for-decomposition">Automaton Diagram for
Decomposition</h3>
<div style="text-align:␣center;">
  <img src="../images/SAR_SUB_DECOMPOSITION.svg" alt="Diagram␣description">
</div>
</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].

Subtraction Strategies: Sliding to Make Bases

Compiled by: Theodore M. Savich

March 30, 2025

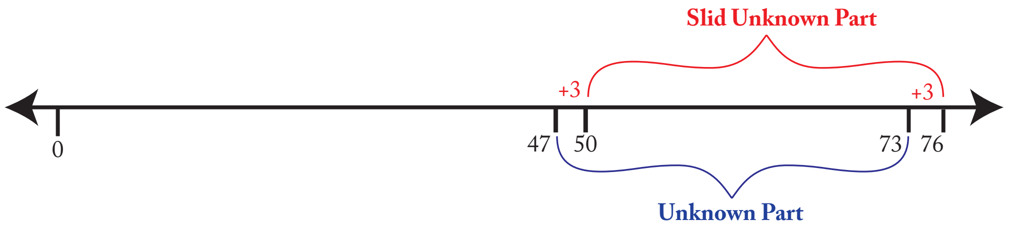
Transcript

Strategy descriptions and examples adapted from Hackenberg (2025). This is not based on a CGI video. I fake a student example.

- Teacher: John had 73 pieces of halloween candy. He gave 47 pieces to his friend. How many pieces of candy does John have left?
- Student: I can pretend I gave away 50 pieces and also pretend I had three more than I did. So that's like 76-50, which is 26.

Notation Representing Rita's Solution:

$$\begin{aligned} 73 - 47 &= \square \\ 73 + 3 &= 76 \\ 47 + 3 &= 50 \\ 73 - 47 &= 76 - 50 \\ &= 26 \end{aligned}$$



In the sliding strategy, you adjust both the number you're subtracting from (the whole) and the number being subtracted (the part) by the same amount. The goal is to shift the subtrahend into a 'friendly' number (usually a multiple of a base). By doing this, the difference between the adjusted values remains identical to the original difference, simplifying the subtraction process.

Description of Strategy

- **Objective:** Adjust both the minuend (known whole) and subtrahend (known part) by the same amount to make the subtraction easier, keeping the difference the same.

Automaton Type

Finite State Automaton (FSA): Adjustments are made consistently and can be tracked without additional memory.

Formal Description of the Automaton

We define the automaton as the tuple

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

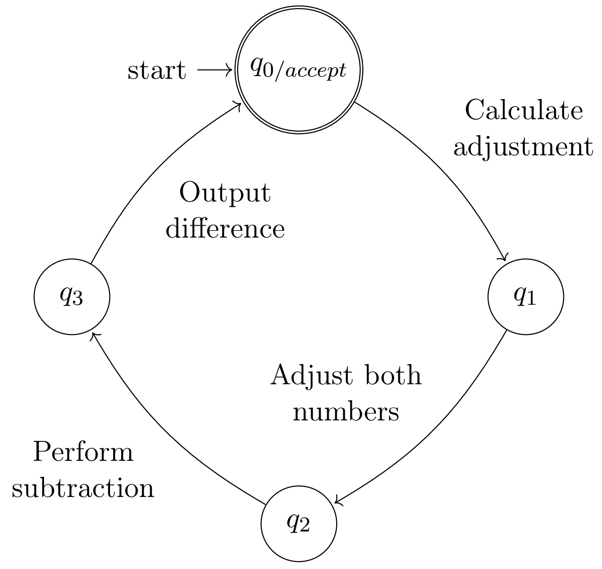
where:

- $Q = \{q_{0/accept},\, q_1,\, q_2,\, q_3\}$ 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.

The transition function δ is defined as follows:

1. $\delta(q_{0/accept},\, \text{“}M, S\text{”}) = q_1$ (Calculate the adjustment needed to make the subtrahend a base multiple.)
2. $\delta(q_1,\, \varepsilon) = q_2$ (Adjust both the minuend and subtrahend by the same amount.)
3. $\delta(q_2,\, \varepsilon) = q_3$ (Perform the subtraction on the adjusted numbers.)
4. $\delta(q_3,\, \varepsilon) = q_{0/accept}$ (Output the final difference.)

Automaton Diagram for Sliding to Make Bases



HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Subtraction Strategies: Sliding to Make Bases</title>
    <style>
        body { font-family: sans-serif; }
        #diagramSlidingSVG { border: 1px solid #d3d3d3; }
        #outputContainer { margin-top: 20px; }
        .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; }
        .original-marker { fill: blue; }
        .adjusted-marker { fill: green; }
        .slide-arrow { fill: none; stroke: darkorange; stroke-width: 1.5; }
        .slide-arrow-head { fill: darkorange; stroke: darkorange; }
        .slide-label { font-size: 10px; fill: darkorange; text-anchor: middle; }
        .difference-bracket { stroke: red; stroke-width: 1.5; fill: none; }
        .difference-label { font-size: 12px; fill: red; text-anchor: middle; }
        .number-line-arrow { fill: black; stroke: black;} /* Arrowhead for the main line
            */
    </style>
</head>
<body>

<h1>Subtraction Strategies: Sliding to Make Bases</h1>

<div>
    <label for="slideMinuend">Minuend:</label>
    <input type="number" id="slideMinuend" value="73">
</div>
<div>
    <label for="slideSubtrahend">Subtrahend:</label>
    <input type="number" id="slideSubtrahend" value="47">
</div>

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

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

<h2>Diagram:</h2>
<svg id="diagramSlidingSVG" width="700" height="300"></svg>

<script>
document.addEventListener('DOMContentLoaded', function() {
    const outputElement = document.getElementById('slidingOutput');
    const minuendInput = document.getElementById('slideMinuend');
    const subtrahendInput = document.getElementById('slideSubtrahend');
    const diagramSVG = document.getElementById('diagramSlidingSVG');

    // --- Helper SVG Functions ---
     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, colorClass = '') { // Added colorClass option
        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 ${colorClass}`.trim()); // Apply
            color class if provided
        tick.setAttribute('stroke', colorClass ? 'currentColor' : 'black'); // Use CSS
            color or default black
        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 createStraightArrow(svg, x1, y1, x2, y2, arrowClass = 'slide-arrow',
        headClass = 'slide-arrow-head', arrowSize = 5) {
        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', arrowClass);
        svg.appendChild(line);

        // Arrowhead pointing right assumed for slide
        const arrowHead = document.createElementNS("http://www.w3.org/2000/svg", 'path');
        arrowHead.setAttribute('d', `M ${x2 - arrowSize} ${y2 - arrowSize/2} L ${x2} ${y2}
             L ${x2 - arrowSize} ${y2 + arrowSize/2} Z`);
        arrowHead.setAttribute('class', headClass);
        svg.appendChild(arrowHead);
    }

    function drawDifferenceBracket(svg, x1, x2, y, label, colorClass = 'difference-') {
        const bracketHeight = 10;
        const path = document.createElementNS("http://www.w3.org/2000/svg", 'path');
        path.setAttribute('d', `M ${x1} ${y - bracketHeight} L ${x1} ${y} L ${x2} ${y} L $
            {x2} ${y - bracketHeight}`);
        path.setAttribute('class', `${colorClass}bracket`);
        svg.appendChild(path);
        createText(svg, (x1 + x2) / 2, y + 15, label, `${colorClass}label`);
    }
    // --- End Helper Functions ---


    // --- Main Sliding Automaton Function ---
    window.runSlidingAutomaton = function() {
        try {
            const minuend = parseInt(minuendInput.value);
            const subtrahend = parseInt(subtrahendInput.value);

            if (isNaN(minuend) || isNaN(subtrahend)) {
               outputElement.textContent = 'Please␣enter␣valid␣numbers␣for␣Minuend␣and␣
                   Subtrahend';
               diagramSVG.innerHTML = ''; return;
            }
             if (subtrahend > minuend) {
                outputElement.textContent = 'Subtrahend␣cannot␣be␣greater␣than␣Minuend.';
                diagramSVG.innerHTML = ''; return;
             }

            let output = `<h2>Sliding to Make Bases</h2>\n\n`;
            output += `<p><strong>Problem:</strong> ${minuend} - ${subtrahend}</p>\n\n`;

            // Calculate adjustment (usually round subtrahend UP)
            const adjustment = (10 - (subtrahend % 10)) % 10;

            const adjustedMinuend = minuend + adjustment;
            const adjustedSubtrahend = subtrahend + adjustment;
            const difference = adjustedMinuend - adjustedSubtrahend; // Should equal
               minuend - subtrahend

            if (adjustment > 0) {
                output += `Step 1: Calculate adjustment to make ${subtrahend} a multiple
                    of 10.\n`;
                output += `<p>Adjustment = +${adjustment}</p>\n`;
                output += `Step 2: Adjust (slide) both numbers by +${adjustment}.\n`
                output += `<p>New Minuend: ${minuend} + ${adjustment} = ${adjustedMinuend
                    }</p>\n`;
                output += `<p>New Subtrahend: ${subtrahend} + ${adjustment} = ${
                    adjustedSubtrahend}</p>\n`;
                output += `Step 3: Subtract adjusted numbers.\n`;
                output += `<p>${adjustedMinuend} - ${adjustedSubtrahend} = ${difference}</
                    p>\n\n`;
            } else {
                output += `Subtrahend ${subtrahend} is already a multiple of 10. No slide
                    needed.\n`;
                output += `<p>Direct Subtraction: ${minuend} - ${subtrahend} = ${
                    difference}</p>\n\n`;
            }


            output += `<strong>Result:</strong> ${difference}`;
            outputElement.innerHTML = output;
            typesetMath();

            // Draw Diagram
            drawSlidingNumberLine(diagramSVG, minuend, subtrahend, adjustedMinuend,
                adjustedSubtrahend, adjustment, difference);

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

    function drawSlidingNumberLine(svg, M, S, M_adj, S_adj, adj, diff) {
         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 * 0.6; // Position number line lower
         const tickHeight = 10;
         const labelOffsetY = 20; // Offset for labels below line
         const slideArrowY = numberLineY - 40; // Y position for slide arrows
         const diffBracketY = numberLineY + 40; // Y position for difference bracket
         const arrowSize = 5;
         const scaleBreakThreshold = 40;

         // Determine range for scaling
         let diagramMin = Math.min(0, S);
         let diagramMax = M_adj; // Need to show the adjusted minuend

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

         if (diagramMin > scaleBreakThreshold) { // Break logic focuses on start
```

```
            displayRangeStart = diagramMin - 10;
            scaleStartX = startX + 30;
            drawScaleBreak = true;
            drawScaleBreakSymbol(svg, scaleStartX - 15, numberLineY);
            drawTick(svg, startX, numberLineY, tickHeight);
            createText(svg, startX, numberLineY + labelOffsetY, '0');
        } else {
            displayRangeStart = 0; // Include 0
            drawTick(svg, startX, numberLineY, tickHeight);
            createText(svg, startX, numberLineY + labelOffsetY, '0');
        }

        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 main line segment
        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
        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);


        // Mark Original Points (Blue)
        const xS = valueToX(S);
        const xM = valueToX(M);
        drawTick(svg, xS, numberLineY, tickHeight, 'original-marker');
        createText(svg, xS, numberLineY + labelOffsetY, S.toString(), 'original-marker');
        drawTick(svg, xM, numberLineY, tickHeight, 'original-marker');
        createText(svg, xM, numberLineY + labelOffsetY, M.toString(), 'original-marker');

        if (adj > 0) { // Only draw adjusted points and arrows if there was a slide
            // Mark Adjusted Points (Green)
            const xS_adj = valueToX(S_adj);
            const xM_adj = valueToX(M_adj);
            drawTick(svg, xS_adj, numberLineY, tickHeight, 'adjusted-marker');
            createText(svg, xS_adj, numberLineY + labelOffsetY + 15, S_adj.toString(), '
                adjusted-marker'); // Offset adjusted label slightly more
            drawTick(svg, xM_adj, numberLineY, tickHeight, 'adjusted-marker');
            createText(svg, xM_adj, numberLineY + labelOffsetY + 15, M_adj.toString(), '
                adjusted-marker'); // Offset adjusted label

            // Draw Slide Arrows (Orange)
            createStraightArrow(svg, xS, slideArrowY, xS_adj, slideArrowY);
            createText(svg, (xS + xS_adj) / 2, slideArrowY - 10, `+${adj}`, 'slide-label'
                );
            createStraightArrow(svg, xM, slideArrowY, xM_adj, slideArrowY);
            createText(svg, (xM + xM_adj) / 2, slideArrowY - 10, `+${adj}`, 'slide-label'
                );

            // Draw Difference Bracket (Red) below adjusted points
             drawDifferenceBracket(svg, xS_adj, xM_adj, diffBracketY, `Difference = ${
                 diff}`);
        } else {
            // Draw Difference Bracket (Red) below original points if no slide
             drawDifferenceBracket(svg, xS, xM, diffBracketY, `Difference = ${diff}`);
        }

    }

    function typesetMath() { /* Placeholder */ }

    // Initial run on page load
    runSlidingAutomaton();

});
</script>

</body>
</html>
```

References

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

Strategic Multiplicative Reasoning - Coordinating Two Counts

Compiled by: Theodore M. Savich

March 30, 2025

Transcript

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

- **Teacher:** Jason has three bags of cookies. There are six cookies in each bag. How many cookies does Jason have altogether?
- **Alex:** There are three bags, right? Six are in each bag. 1, 2, 3, 4, 5, 6. 1, 2, 3, 4, 5, 6. 1, 2, 3, 4, 5, 6. 1, 2, 3, 4, 5, 6, will go in this bag. 1, 2, 3, 4, 5, 6. Six will go into this bag. And 1, 2, 3, 4, 5, 6, will go into this bag. So 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18. Eighteen cookies are in each bag
- **Teacher:** Nice, thank you. Put those aside.

Alex started by arranging three unifix cubes. Soon, he realized that he needed to count cookies. He initially counted in groups of six cubes, even exceeding three complete groups. Recognizing this approach was inefficient, he began again—this time, he placed one cube to represent a bag and then added six cubes to stand for the cookies that would fill that bag. He repeated this process three times. Finally, by counting all the cubes (each standing in for a cookie), he determined there were 18 cookies in total.

In general, count incrementally by ones, but keep track of how many groups you are counting to coordinate the two distinct types of units involved.

Coordinating Two Counts by Ones (C2C)

Description of Strategy:

- **Objective:** Count the total number of items by counting each item one by one, while keeping track of both the number of groups and the number of items in each group.
- **Method:** For each group, count the items in that group by ones, and repeat this for each group, incrementing the total count.

Automaton Type:

Finite State Automaton (FSA) with counters.

Formal Description of the Automaton

We define the automaton as the tuple

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

where:

- $Q = \{q_{0/accept}, q_{\text{count\_items}}, q_{\text{next\_group}}\}$ is the set of states.
- Σ is the input alphabet (used, for example, to read the initial values for the problem).
- $q_{0/accept}$ is the start state, which is also the accept state.
- $F = \{q_{0/accept}\}$ is the set of accepting states.
- $V = \{$GroupCounter (G), ItemCounter (I), TotalCounter (T), GroupSize (S), TotalGroups (N)$\}$ is the set of variables.

Key Transitions:

1. **Initialization:** From $q_{0/accept}$, on reading the input (e.g., the values of S and N), set $G = 0$, $I = 0$, and $T = 0$, then move to $q_{\text{count\_items}}$.
2. **Counting Items:** In $q_{\text{count\_items}}$, for each item in the current group, increment I and T (looping until $I = S$).
3. **Moving to Next Group:** When $I = S$ (the current group is complete), transition to $q_{\text{next\_group}}$ where G is incremented and I is reset to 0.
4. **Completion:** In $q_{\text{next\_group}}$, if $G = N$ (all groups have been counted), transition back to $q_{0/accept}$ to output the total count T; otherwise, return to $q_{\text{count\_items}}$ for the next group.

Automaton Diagram for C2C

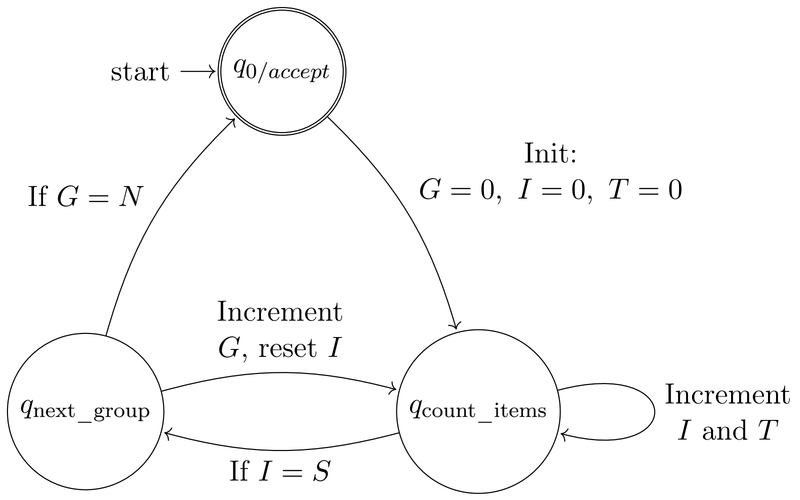


Figure 1: FSA with counters to coordinate item and group counting by ones.

Extending to a Two-Stack Automaton (2-PDA)

While the above FSA captures the essence of coordinating two types of counts (items and groups), it does not explicitly illustrate the use of a stack. If one requires *unbounded* counting or more advanced structure (e.g., repeated addition for multiplication in a more formal sense), a single-stack PDA can be designed. However, to **compose two distinct PDAs**—one for the item count and one for the group count—and retain each one's push/pop operations, we can move to a **two-stack pushdown automaton (2-PDA)**. This sort of machine:

- Uses two independent stacks, Stack_1 and Stack_2, each manipulated by transitions in its own sub-automaton.
- Has states that combine the "local states" of the separate PDAs. A state in the 2-PDA is effectively a pair (q_1, q_2), where q_1 is from the item-counting PDA and q_2 is from the group-counting PDA.
- Pushes and pops symbols from either (or both) stacks, depending on which sub-automaton's transition is activated.

Formal 2-PDA Composition

Let:

$$P_1 = (Q_1,\ \Sigma,\ \Gamma_1,\ \delta_1,\ q_{1,0},\ F_1) \quad \text{and} \quad P_2 = (Q_2,\ \Sigma,\ \Gamma_2,\ \delta_2,\ q_{2,0},\ F_2)$$

be two PDAs (each with its own stack alphabet, Γ_1 and Γ_2, and transition functions δ_1 and δ_2). The **two-stack automaton** $P_\times$ that composes them is:

$$P_\times = \big(Q_1 \times Q_2,\ \Sigma,\ \Gamma_1,\ \Gamma_2,\ \delta_\times,\ (q_{1,0}, q_{2,0}),\ F_1 \times F_2\big),$$

where

$$\delta_\times\big((q_1, q_2),\ a,\ X,\ Y\big) = \big\{\big((q_1', q_2'),\ \alpha,\ \beta\big) \,\big|\, (q_1', \alpha) \in \delta_1(q_1,\ a,\ X) \text{ and } (q_2', \beta) \in \delta_2(q_2,\ \epsilon,\ Y)\big\},$$

and similarly for transitions where P_2 processes input a while P_1 processes ϵ. The notation means:

- On input symbol a, with the top of Stack_1 being X and the top of Stack_2 being Y, the composite automaton transitions to (q_1', q_2').
- It replaces X with α in Stack_1 (possibly pushing or popping multiple symbols) and Y with β in Stack_2.

Interpreting the Two Stacks for Multiplication - **Stack_1**: Manages the state of counting items in one group (similar to your single-stack counting idea, but restricted to item-level detail). - **Stack_2**: Manages the state of counting how many groups have been multiplied so far (e.g., for repeated addition).

During each "repeated addition" cycle: 1. The item-counting sub-automaton (PDA_1) increments the partial total by the group size, pushing/popping from Stack_1. 2. The group-counting sub-automaton (PDA_2) tracks how many times this addition has been done, pushing/popping from Stack_2.

Once PDA_2 indicates all groups have been accounted for, the 2-PDA halts or transitions to an accepting state.

Example of Counting Three Groups of Six (High-Level 2-PDA)

1. **Stacks Initialization:**
   - Stack_1 starts with the necessary markers/symbols to begin item counting.
   - Stack_2 starts with a symbolic representation of how many groups remain (e.g., 3).
2. **Item Counting Process (Stack_1):**
   - Each time the automaton processes the addition of 6 items to the partial total, it pushes/pops in Stack_1 to record digits in base-b or some other scheme.
3. **Group Countdown (Stack_2):**
   - After finishing one addition cycle for 6 items, pop one "group token" from Stack_2.
   - If Stack_2 is not empty, move on to add another 6.
   - If Stack_2 becomes empty, the multiplication is complete.

Why a 2-PDA? Composing two separate single-stack PDAs *in parallel* effectively yields a machine with two stacks. The 2-PDA formalism lets each "sub-automaton" maintain its independent pushdown memory, which can be advantageous if you conceptually want to keep the logic of item-counting and group-counting separate. In theoretical terms, a 2-PDA is already as powerful as a Turing machine, so it can handle the entire repeated-addition multiplication process without additional resources.

Conclusion on the Two-Stack Approach

Using a two-stack automaton is a straightforward way to **combine** two independently designed PDAs so that each retains its own stack-based memory management. This might be done for instructional clarity or for theoretical completeness when demonstrating that distinct counting mechanisms can be kept separate. In practice, a single-stack PDA can also implement multiplication by carefully interleaving the logic in one stack. However, splitting the tasks across two separate stacks can simplify the conceptual breakdown of item counting versus group counting.

HTML Implementation

```html
<!DOCTYPE html>
<html>
<head>
    <title>Multiplication: Coordinating Two Counts by Ones (C2C)</title>
    <style>
        body { font-family: sans-serif; line-height: 1.6; }
        .representation-section { margin-bottom: 20px; padding: 10px; border: 1px solid #
            eee; min-height: 50px;}
        .control-section { margin-bottom: 20px; }
        label { margin-right: 5px;}
        input[type=number] { width: 60px; margin-right: 15px;}
        .box { /* Style for individual item box */
            display: inline-block;
            width: 15px; height: 15px; margin: 1px;
            background-color: lightblue; border: 1px solid #666;
            vertical-align: middle;
        }
        .tally-mark { /* Style for group tally */
            font-family: monospace;
            font-size: 24px;
            margin-right: 4px; /* Spacing between tallies */
            display: inline-block;
            vertical-align: middle;
            color: darkgreen;
        }
         .group-spacer { /* Visual space between groups of boxes */
             display: inline-block;
             width: 10px;
             height: 15px;
             vertical-align: middle;
         }
        button { padding: 5px 10px; font-size: 1em; margin-right: 5px; }
        #numericValue { font-size: 1.5em; font-weight: bold; color: darkblue; }
        #statusMessage { color: red; font-weight: bold; }

    </style>
</head>
<body>

    <h1>Strategic Multiplicative Reasoning - Coordinating Two Counts by Ones (C2C)</h1>

    <div class="control-section">
        <label for="groupSizeInput">Group Size (S):</label>
        <input type="number" id="groupSizeInput" value="6" min="1">
        <label for="numGroupsInput">Number of Groups (N):</label>
        <input type="number" id="numGroupsInput" value="3" min="1">
        <button onclick="resetSimulation()">Start/Reset</button>
        <button onclick="countNextItem()" id="incrementBtn">Count Next Item</button>
         <span id="statusMessage"></span>
    </div>

    <p><strong>Total Items Counted:</strong> <span id="numericValue">0</span></p>

    <div class="representation-section">
        <strong>Groups Tracked (Tallies represent completed groups):</strong><br />
        <span id="tallyDisplay"></span>
    </div>

    <div class="representation-section">
        <strong>Items Counted (Boxes grouped by Group Size):</strong><br />
        <span id="boxesDisplay"></span>
    </div>


    <script>
        // --- Simulation State Variables ---
        let groupSize = 6;
        let numGroups = 3;
        let currentGroupNum = 0; // How many groups *completed*
        let currentItemInGroup = 0; // How many items counted *in the current group*
        let currentTotalCount = 0; // Total items overall
        let isComplete = true; // Start in a non-counting state

        // --- DOM Element References ---
        const numericValueSpan = document.getElementById("numericValue");
        const boxesContainer = document.getElementById("boxesDisplay");
        const tallyContainer = document.getElementById("tallyDisplay");
        const incrementBtn = document.getElementById("incrementBtn");
        const statusMessage = document.getElementById("statusMessage");
        const groupSizeInput = document.getElementById("groupSizeInput");
        const numGroupsInput = document.getElementById("numGroupsInput");

        // --- Simulation Functions ---
        function resetSimulation() {
            groupSize = parseInt(groupSizeInput.value) || 1; // Ensure at least 1
            numGroups = parseInt(numGroupsInput.value) || 1; // Ensure at least 1
            groupSizeInput.value = groupSize; // Update input in case of default
            numGroupsInput.value = numGroups;

            currentGroupNum = 0;
            currentItemInGroup = 0;
            currentTotalCount = 0;
            isComplete = (numGroups <= 0 || groupSize <= 0); // Complete if invalid input

            updateDisplay();
            statusMessage.textContent = isComplete ? "Set Group Size and Num Groups > 0, 
                then Reset." : "Ready to count.";
        }

        function countNextItem() {
            if (isComplete) {
                statusMessage.textContent = "Counting complete! Press Reset to start again.
                    ";
                return;
            }

            statusMessage.textContent = ""; // Clear message

            // Increment total count (State q_count_items: Increment T)
            currentTotalCount++;

            // Increment item within the current group (State q_count_items: Increment I)
            currentItemInGroup++;

            // Check if current group is finished (State q_count_items -> q_next_group
                transition check: I == S?)
            if (currentItemInGroup === groupSize) {
                currentGroupNum++; // Increment completed group count (Action: G = G + 1)
                currentItemInGroup = 0; // Reset item count for next group (Action: I = 0)

                // Check if all groups are finished (State q_next_group -> q0/accept check:
                     G == N?)
                if (currentGroupNum === numGroups) {
                    isComplete = true; // All groups done
                    statusMessage.textContent = "Counting␣complete!";
                } else {
                    // Transition back to q_count_items conceptually for the next group
                    statusMessage.textContent = `Finished Group ${currentGroupNum}.
                        Starting Group ${currentGroupNum + 1}...`;
                }
            } else {
                 statusMessage.textContent = `Counting item ${currentItemInGroup} in Group
                     ${currentGroupNum + 1}...`;
            }


            updateDisplay();
        }


        function updateDisplay() {
            // Update numeric display
            numericValueSpan.textContent = currentTotalCount;

            // Enable/Disable Increment Button
            incrementBtn.disabled = isComplete;

            // --- Update Tallies (Groups Tracked) ---
            tallyContainer.innerHTML = ""; // Clear previous
            // Draw one tally for each *completed* group
            tallyContainer.textContent = "|".repeat(currentGroupNum);
            tallyContainer.className = 'tally-mark'; // Apply class


            // --- Update Boxes (Items Counted) ---
            boxesContainer.innerHTML = ""; // Clear previous
            for (let i = 1; i <= currentTotalCount; i++) {
                 const box = document.createElement("div");
                 box.className = "box";
                 boxesContainer.appendChild(box);

                // Add a visual spacer after each completed group (except the last item)
                if (i % groupSize === 0 && i < currentTotalCount) {
                    const spacer = document.createElement("span");
                    spacer.className = "group-spacer";
                    boxesContainer.appendChild(spacer);
                }
            }

        } // End of updateDisplay

        // Initialize the display on page load
        resetSimulation(); // Start with defaults loaded

    </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].

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

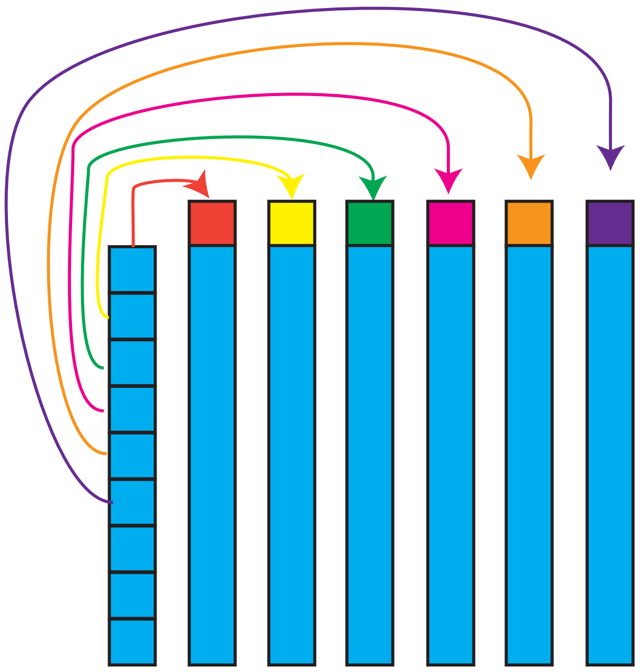
Compiled by: Theodore M. Savich

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

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

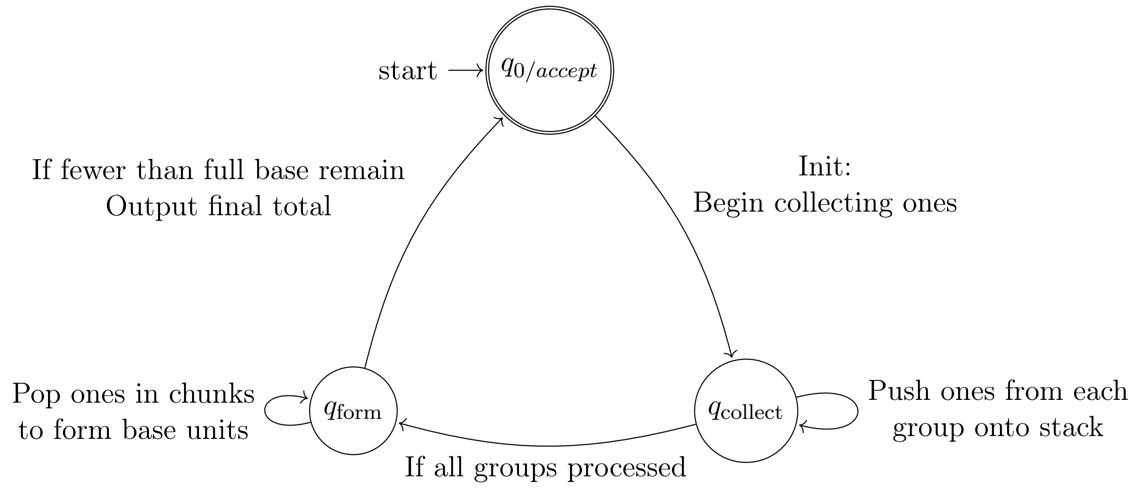
where:

- $Q = \{q_{0/accept}, q_{\text{collect}}, q_{\text{form}}\}$ 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

```html
<!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);

                if (isNaN(numGroups) || isNaN(itemsPerGroup) || numGroups <= 0 || itemsPerGroup <= 0) {
                    outputElement.textContent = "Please enter valid positive numbers";
                    diagramSVG.innerHTML = ''; return;
                }

                const totalItems = numGroups * itemsPerGroup;
                const finalTensCount = Math.floor(totalItems / 10);
                const finalOnesCount = totalItems % 10;
                const numGroupsWord = numberToWord(numGroups); // Get word for groups

                // --- Generate Text Notation (Matching PDF) ---
                let output = `<h2>Conversion to Bases and Ones (CBO) - Notation</h2>\n\n`;
                output += `<p class="notation-line problem">${numGroupsWord}  ${itemsPerGroup} = ?</p>\n`;

                if (itemsPerGroup < 10 && numGroups > 1) {
                    const neededPerGroup = 10 - itemsPerGroup;
                    const groupsToComplete = numGroups - 1; // Try to complete all but one group
                    const totalNeeded = groupsToComplete * neededPerGroup;
                    // Find how many ones are left in the last group after donating
                    const onesLeftInLastGroup = itemsPerGroup - totalNeeded;

                    if (onesLeftInLastGroup >= 0) { // Check if the last group had enough
                         output += `<p class="notation-line">= ${numberToWord(groupsToComplete)}  ${itemsPerGroup} + ${itemsPerGroup}</p>\n`;
                         output += `<p class="notation-line">= ${numberToWord(groupsToComplete)}  ${itemsPerGroup} + ${totalNeeded} + ${onesLeftInLastGroup}</p>\n`; // Show split of last group
                         output += `<p class="notation-line">= ${numberToWord(groupsToComplete)}  (${itemsPerGroup} + ${neededPerGroup}) + ${onesLeftInLastGroup}</p>\n`; // Show distribution
                         output += `<p class="notation-line">= ${numberToWord(groupsToComplete)}  10 + ${onesLeftInLastGroup}</p>\n`;
                         output += `<p class="notation-line">= ${groupsToComplete * 10} + ${onesLeftInLastGroup}</p>\n`;
                         output += `<p class="notation-line">= ${totalItems}</p>\n`;
                    } else {
                        // Logic for needing more than one group to decompose is more complex
                        // For simplicity, just show the direct calculation result for text if simple decomp fails
                         output += `<p class="notation-line">= ${totalItems} (Direct Calculation)</p>\n`;
                    }
                } else {
                    // If itemsPerGroup >= 10 or only one group, direct calculation is simpler notation
                     output += `<p class="notation-line">= ${totalItems} (Direct Calculation)</p>\n`;
                }


```

```
                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 });
                     itemYOffset += blockUnitSize + blockSpacing;
                 }
                 currentX = groupStartX + blockUnitSize + groupSpacingX; // Next group
                     starts after one block width + spacing
                 section1MaxY = Math.max(section1MaxY, currentY + itemYOffset);
             }

              // --- 2. Redistribution Arrows (Conceptual) ---
              // Only draw if redistribution is feasible (S<10, N>1, and last group has
                  enough)
              const neededPerGroup = (itemsPerGroup < 10) ? 10 - itemsPerGroup : 0;
              const groupsToComplete = numGroups - 1;
              const totalNeeded = groupsToComplete * neededPerGroup;
              const onesLeftInLastGroup = itemsPerGroup - totalNeeded;

              if (neededPerGroup > 0 && onesLeftInLastGroup >= 0 && numGroups > 1) {
                  // Find blocks in the last group to be the source
                  let sourceBlocks = initialGroupsData.filter(d => d.group === numGroups -
                      1).slice(0, totalNeeded); // Get the first 'totalNeeded' blocks from
                      the last group
                  let targetGroups = initialGroupsData.filter(d => d.group < numGroups - 1)
                      ;

                  let sourceIndex = 0;
                  for (let g = 0; g < groupsToComplete; g++) {
                       // Find the top-most block of the target group 'g'
                       let targetBlock = targetGroups.find(d => d.group === g && d.item ===
                            itemsPerGroup -1); // Top item in the target group
                       if (targetBlock && sourceIndex < sourceBlocks.length) {
                           let sourceBlock = sourceBlocks[sourceIndex];
                            // Draw arrow from source block to above target block
                            createCurvedArrow(svg,
                                sourceBlock.cx, sourceBlock.cy, // Start center of source
                                    block
                                targetBlock.cx, targetBlock.y + arrowOffsetY, // End
                                    slightly above target block
                                (sourceBlock.cx + targetBlock.cx) / 2, sourceBlock.cy +
                                    arrowControlOffsetY // Control point for arc
                            );
                           sourceIndex++;
                       }
                       // We need to distribute 'neededPerGroup' to each target group
                       // This loop just draws one arrow per target group for simplicity
                       // A more complex viz could draw neededPerGroup arrows per target
                           group
                  }
              }


             currentY = section1MaxY + sectionSpacingY;


             // --- 3. Final Result Visualization (Base-10 Blocks) ---
             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].

Multiplication Strategies: Commutative Reasoning

Compiled by: Theodore M. Savich

April 1, 2025

Commutative Action for Multiplication

Imagine a situation where we have six chocolate chip cookies with 4 chocolate chips in each cookie. That's 24 chocolate chips. Instead, we imagine we have four chocolate chip cookies, and each cookie has 6 chocolate chips. That's still 24 chocolate chips, but not enough cookies to feed 4 kids! The commutative property of multiplication,

- Definition: For any two natural numbers a and b,

$$a \quad b = b \quad a.$$

- Example: $3 \quad 4 = 4 \quad 3$.

is fine for purely abstract mathematical contexts, but in *equal groups multiplication problems* – the sort of problems that most people encounter when learning about multiplication for the first time – the order of the factors can make a big difference.

However, there is a big difference between recognizing the commutative property holds for the number of chocolate chips and the fact that you would have two crying kids if there were 6 kids and you only had 4 cookies.

For equal groups multiplication:

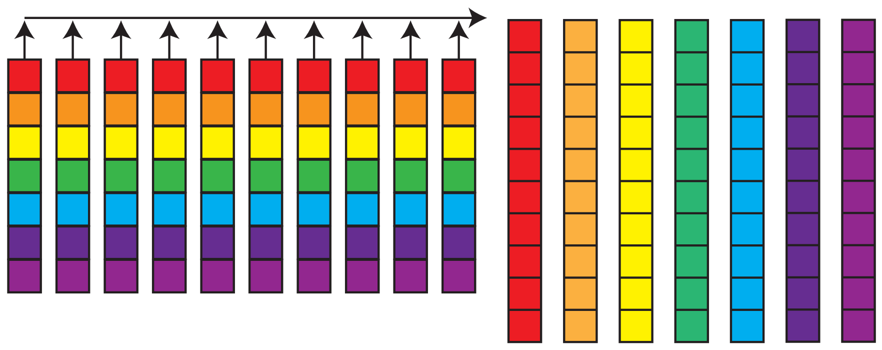
$$\boxed{\text{number of groups}} \quad \boxed{\text{number of items in each group}} = \boxed{\text{total number of items}}$$

To act (or reason) commutatively, the number of items in a group needs to be repackaged as the number of groups, while the number of groups transforms into the number of items in each group.

Definition and Example

Imagine this situation, from Hackenberg (2025): we have ten packages of rainbow flavored candies, and each package contains the 7 candies, one of each of the 'colors of the rainbow': red, orange, yellow, green, blue, indigo, and violet.

It can be hard to count 7 objects ten times. 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 is a lot of work! We can *repackage* the candies into seven packages of 10 candies each - all the reds together, all the oranges together, and so on. The result is seven 10s, which is a lot easier to count: 10 + 10 + 10 + 10 + 10 + 10 + 10 = 70.



Objective of the Automaton

- Input: A multiplication expression a b.
- Output: The transformed expression b a.
- Functionality: Recognize when a multiplication expression is presented and apply the commutative property to reorder the operands.

Automaton Type Selection

Finite State Transducer (FST)

- Transduction Capability: Unlike finite state automata (FSA) that merely recognize languages, an FST can transform input strings into output strings.
- Suitability: Ideal for tasks involving input-output transformations, such as repackaging operands in a multiplication expression.

Designing the FST for Commutative Reasoning

Components of the FST

1. States (Q):
   - q_0: Start state.
   - q_1: Reading the first operand.
   - q_2: Reading the multiplication symbol ( ).
   - q_3: Reading the second operand.
   - q_4: Applying the commutative transformation.
   - q_{accept}: Accepting state; transformation complete.
2. Input Alphabet ( ):
   - Digits: $f0; 1; 2; 3; 4; 5; 6; 7; 8; 9g$
   - Multiplication symbol:
3. Output Alphabet ( ):
   - Digits: $f0; 1; 2; 3; 4; 5; 6; 7; 8; 9g$

- Multiplication symbol:

4. Transition Function ( ): Defines how the FST transitions between states based on input symbols and produces corresponding output symbols.
5. Start State: q_0
6. Accepting State: q_{accept}

Transition Function Details (Single-Digit Operands)

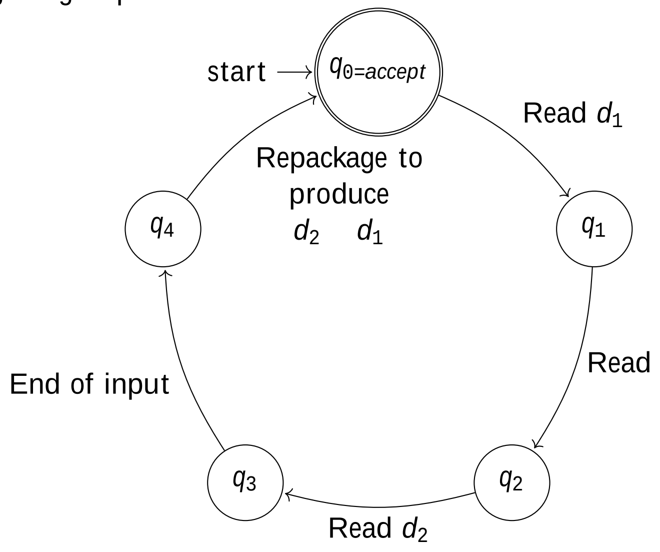
For simplicity, assume operands are single digits. The FST behaves as follows:

Current State	Input Symbol	Read Symbol	Next State	Output Symbol
q_0	Any digit d_1	d_1	q_1	d_1
q_1			q_2	
q_2	Any digit d_2	d_2	q_3	d_2
q_3	End of input	—	q_4	—
q_4	—	—	q_{accept}	Output repackaged expression: $d_2 \quad d_1$

Automaton Diagrams

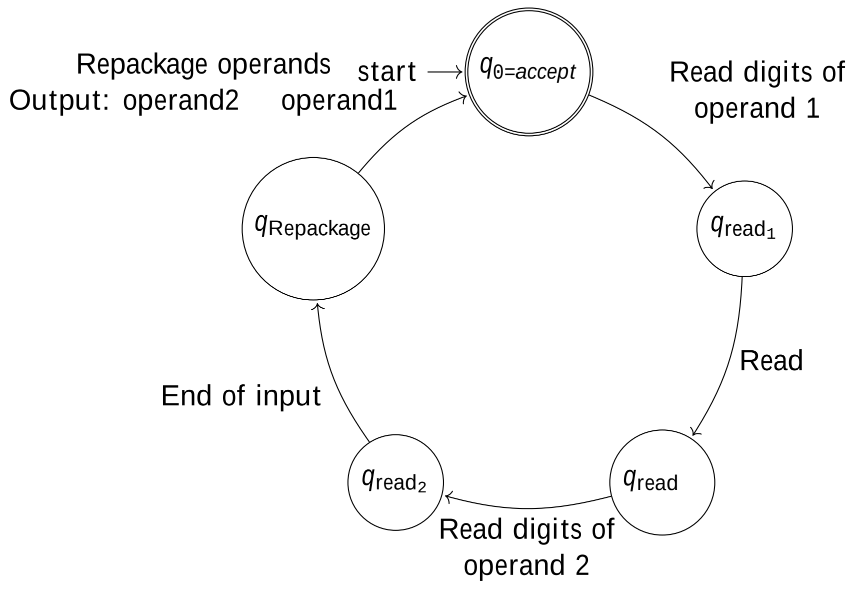
Circular Diagram for Single-Digit Operands

Below is the circular state diagram for the FST (with the start and accept states merged) for single-digit operands.



Circular Diagram for Multi-Digit Operands

For multi-digit operands, the FST bu□ers digits until the entire operand is read, then repackages the operands. The following circular diagram represents an enhanced FST:



Example Execution

Problem:

3   4

Execution Steps:

1. q_0: Reads the digit ‘3’, outputs ‘3’, then moves to q_1.
2. q_1: Reads ‘   ’, outputs ‘   ’, then moves to q_2.
3. q_2: Reads the digit ‘4’, outputs ‘4’, then moves to q_3.
4. q_3: End of input is detected; transition to q_4.
5. q_4: Repackages the operands to produce ‘4   3’, and transitions back to $q_{0=accept}$ (accepting state).

Output:

4   3

Conclusion

By designing this Finite State Transducer (FST), we e□ectively model the commutative property of multiplication as a transformation process. The single-digit version demonstrates the basic concept, while the multi-digit version shows how the automaton can be extended to handle more complex expressions by bu□ering entire operands before applying the repackage.

HTML Implementation

```html
<!DOCTYPE html>
<html>
<head>
    <title>Commutative Multiplication</title>
<style>
    body { font-family: sans-serif; }
    .cube-row { display: flex; margin-bottom: 2px; } /* Arrange cubes in a row */
    .cube {
        width: 15px; /* Cube size */
        height: 15px;
        border: 1px solid #ccc; /* Cube border */
        margin-right: 2px; /* Spacing between cubes */
        display: inline-block; /* Ensure inline display for flexbox */
    }
    /* Rainbow colors for cubes - you can customize these */
    .cube.red { background-color: red; }
    .cube.orange { background-color: orange; }
    .cube.yellow { background-color: yellow; }
    .cube.green { background-color: green; }
    .cube.blue { background-color: blue; }
    .cube.indigo { background-color: indigo; }
    .cube.violet { background-color: violet; }

</style>
</head>
<body>
    <h1>Commutative Reasoning for Multiplication</h1>

    <div>
        <label for="commuteA">Factor 1:</label>
        <input type="number" id="commuteA" value="10">
    </div>
    <div>
        <label for="commuteB">Factor 2:</label>
        <input type="number" id="commuteB" value="7">
    </div>

    <button onclick="runCommutativeAutomaton()">Repackage and Visualize</button>

    <div id="commuteOutput">
        <!-- Output will be displayed here -->
    </div>

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

    <script>
        function openPdfViewer() {
            // Opens the PDF documentation for the strategy.
            window.open('../SMR_MULT_Commutative_Reasoning.pdf', '_blank');
        }
    </script>

    <script>
        document.addEventListener(’DOMContentLoaded’, function() {
            const commuteOutputElement = document.getElementById(’commuteOutput’);
            const commuteAInput = document.getElementById(’commuteA’);
            const commuteBInput = document.getElementById(’commuteB’);

            window.runCommutativeAutomaton = function() {
                try {
                    const factorA = commuteAInput.value;
                    const factorB = commuteBInput.value;

                    if (isNaN(parseInt(factorA)) || isNaN(parseInt(factorB)) || parseInt(factorA) <= 0 || parseInt(factorB) <= 0) {
                        commuteOutputElement.textContent = "Please␣enter␣valid␣positive␣numbers␣for␣both␣factors";
                        return;
                    }

                    let output = ’’;
                    output += ‘<h2>Commutative Repackaging for Multiplication</h2>\n\n‘;
                    output += ‘<p><strong>Original Expression:</strong> ${factorA} &times; ${factorB}</p>\n‘; // Updated to display the multiplication symbol correctly

                    // --- Simulate FST Transformation ---
                    const transformedFactorA = factorB;
                    const transformedFactorB = factorA;

                    output += ‘<p><strong>Applying Commutative Repackaging...</strong></p>\n‘;
                    output += ‘<p>We transform the expression by swapping the order of the factors.</p>\n‘;
                    output += ‘<p><strong>Repackaged Expression:</strong> ${transformedFactorA} &times; ${transformedFactorB}</p>\n\n‘;

                    // --- Visualize with Colorful Cubes ---
                    const numFactorA = parseInt(factorA);
                    const numFactorB = parseInt(factorB);
                    const productAB = numFactorA * numFactorB;
                    const productBA = parseInt(transformedFactorA) * parseInt(transformedFactorB);

                    output += ‘<p><strong>Visualizing the Repackaging:</strong></p>\n‘;

                    // Arrangement 1 (Original: A x B) - Cubes
                    output += ‘<p><strong>Arrangement 1: ${factorA} groups of ${factorB} items each</strong></p>\n‘;
                    output += ‘<p>Visual representation:</p>\n‘;
                    for (let i = 0; i < numFactorA; i++) {
                        output += ‘<div class=’cube-row’>‘; // Start a new row for cubes
                        for (let j = 0; j < numFactorB; j++) {
                            const rainbowColors = [’red’, ’orange’, ’yellow’, ’green’, ’blue’, ’indigo’, ’violet’];
                            const colorClass = rainbowColors[j % rainbowColors.length]; // Cycle through rainbow colors
                            output += ‘<span class=’cube␣${colorClass}’></span>‘; // Create a cube with color class
                        }
                        output += ‘</div>‘; // End the cube row
                    }
                    output += ‘<p>Total: ${productAB} items</p>\n\n‘;

                    // Arrangement 2 (Repackaged: B x A) - Cubes
                    output += ‘<p><strong>Arrangement 2: ${transformedFactorA} groups of ${transformedFactorB} items each</strong></p>\n‘;
                    output += ‘<p>Visual representation:</p>\n‘;
                    for (let i = 0; i < parseInt(transformedFactorA); i++) {
                        output += ‘<div class=’cube-row’>‘; // Start a new row
                        for (let j = 0; j < parseInt(transformedFactorB); j++) {
                            const rainbowColors = [’red’, ’orange’, ’yellow’, ’green’, ’blue’, ’indigo’, ’violet’];
                            const colorClass = rainbowColors[j % rainbowColors.length];
                            output += ‘<span class=’cube␣${colorClass}’></span>‘; // Create colored cube
                        }
                        output += ‘</div>‘; // End row
                    }
                    output += ‘<p>Total: ${productBA} items</p>\n\n‘;


                    output += ‘<p><strong>Conclusion:</strong></p>\n‘;
                    output += ‘<p>By commutatively repackaging ${factorA} &times; ${factorB} into ${transformedFactorA} &times; ${transformedFactorB}, we change the grouping but maintain the same total quantity (${productAB} = ${productBA}).</p>\n‘;


                    commuteOutputElement.innerHTML = output;


                } catch (error) {
                    commuteOutputElement.textContent = ‘Error: ${error.message}‘;
                }
            };
        });
    </script>
</body>
</html>
```

Multiplication Strategies: Doubling

Theodore M. Savich

March 9, 2025

Doubling

Description of Strategy:

- **Objective:** Use doubling to quickly reach the total number of items by doubling group sizes or totals.
- **Method:** Double the number of items (and the number of groups) repeatedly until reaching or surpassing the target total, then adjust as needed.

Automaton Type:

Finite State Automaton with Registers (Counters): Counters are used to track the current total and the number of groups.

Formal Description of the Automaton

We define the automaton as the tuple

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

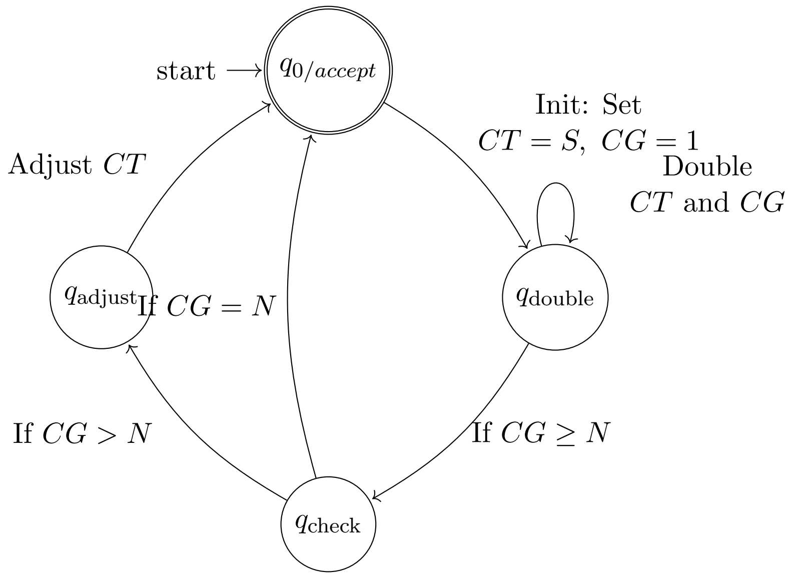
where:

- $Q = \{q_{0/accept}, q_{\text{double}}, q_{\text{check}}, q_{\text{adjust}}\}$ is the set of states. Here, $q_{0/accept}$ serves as both the start and accept state.
- Σ is the input alphabet (used to initialize the problem parameters).
- $F = \{q_{0/accept}\}$ is the set of accepting states.
- $V = \{\text{CurrentTotal (CT), CurrentGroups (CG), GroupSize (S), TotalGroups (N)}\}$ is the set of registers.

The key transitions are as follows:

1. **Initialization:** From $q_{0/accept}$, on reading the input values (with S and N), initialize $CT \leftarrow S$ and $CG \leftarrow 1$, then transition to q_{double}.
2. **Doubling:** In q_{double}, repeatedly double both CT and CG (i.e., update $CT \leftarrow 2 \times CT$ and $CG \leftarrow 2 \times CG$) until $CG \geq N$.
3. **Checking:** In q_{check}, if $CG = N$ then the target total is reached, and the automaton transitions to the accept state. If $CG > N$, transition to q_{adjust} to fine-tune CT.
4. **Adjustment:** In q_{adjust}, adjust CT appropriately (e.g., subtract the excess) before outputting the final total.

Automaton Diagram for Doubling



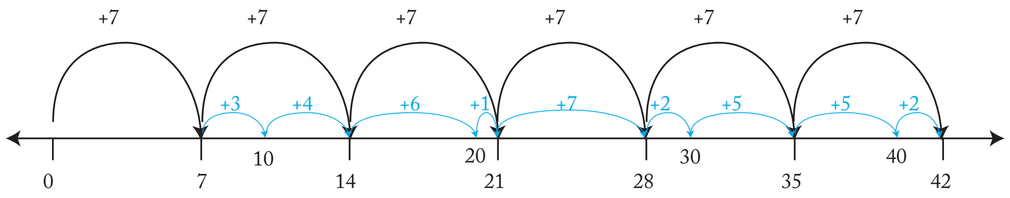
Multiplication Strategies: Strategic Counting

Theodore M. Savich

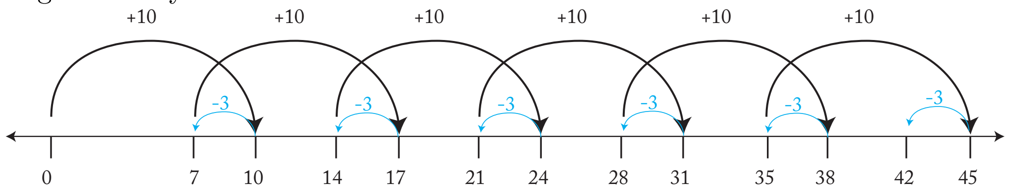
Strategy descriptions and examples adapted from Hackenberg (2025).

This method uses additive techniques—like Rearranging Ones to Make Bases (RMB), Chunking, or Rounding—to manage two distinct types of units while calculating the total number of items. The goal isn't to count one by one but to employ a more efficient grouping strategy. Nonetheless, each group is still added sequentially, one at a time.

For example, if you have six groups of 7, you could use rearranging to make bases several times – keeping track of the number of groups and the total number of items in each group – to obtain 42.



Or, for the same problem, you could pretend to add 10, then adjust back by three, over and over again until you reach the total.



Description of Strategy

- **Objective:** Use any of several **additive** strategies (for example, rearranging ones to make bases, chunking, rounding, etc.) to add the group size without counting each item by ones.
- **Method:** Instead of incrementing one by one, interpret the multiplication problem as repeated addition of the group size, then apply one of the **efficient** addition strategies for each step.

Automaton Type

Finite State Automaton with Registers (Counters). Below is a high-level representation. A two-stack automaton approach is described later.

Formal Description of a High-Level FSA

We define the automaton as:

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

where:

- $Q = \{q_{0/accept}, q_{\text{add\_group}}, q_{\text{next\_group}}\}$.
- $q_{0/accept}$ is both the start and accept state.
- $F = \{q_{0/accept}\}$.
- $V = \{\text{GroupCounter}(G),\ \text{Total}(T),\ \text{GroupSize}(S),\ \text{TotalGroups}(N)\}$.

1. **Initialization:** From $q_{0/accept}$, read S and N. Set $G = 0$ and $T = 0$, then transition to $q_{\text{add\_group}}$.
2. **Add the Group Size:** In $q_{\text{add\_group}}$, add S to T. (This step uses a chosen addition strategy like chunking or rearranging.)
3. **Next Group:** If $G < N$, transition to $q_{\text{next\_group}}$, increment G, and return to $q_{\text{add\_group}}$. If $G = N$, move back to $q_{0/accept}$.

Automaton Diagram for Strategic Counting (High-Level)

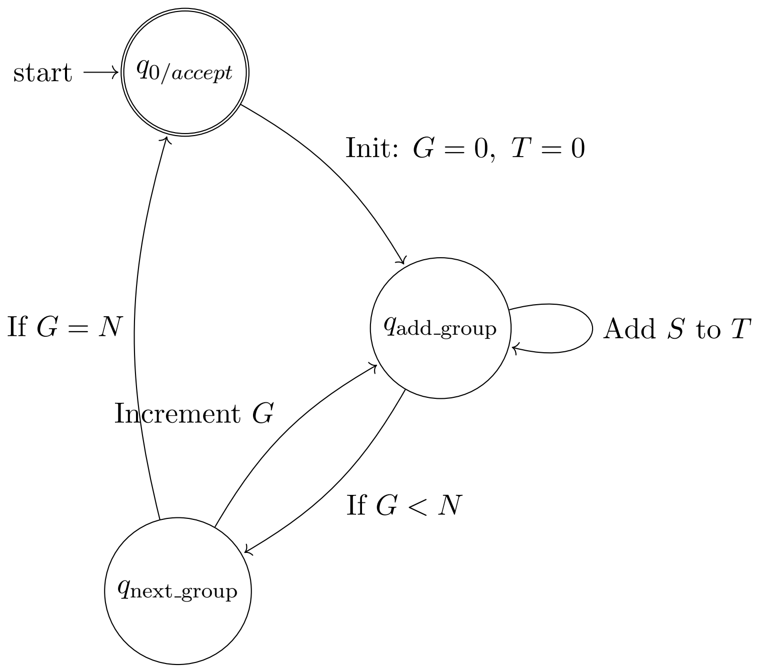


Figure 1: High-level FSA for Multiplying via Strategic Counting

Two-Stack Automaton (2-PDA) for Strategic Counting

Rather than a single FSA, we can compose two distinct Pushdown Automata:

- **Sub-PDA**$_1$: Manages how many groups are left to process.
- **Sub-PDA**$_2$: Implements one of the sophisticated addition strategies for adding the group size to the running total.

A single-stack PDA cannot hold both sub-automata memories separately. Therefore, we move to a **two-stack automaton** (2-PDA), which formally:

$$P_\times = (Q_1 \times Q_2,\ \Sigma,\ \Gamma_1,\ \Gamma_2,\ \delta_\times,\ (q_{1,0}, q_{2,0}),\ F_1 \times F_2).$$

Here, Γ_1 is the stack alphabet for the group-counting sub-PDA, and Γ_2 is the alphabet for the addition-strategy sub-PDA.

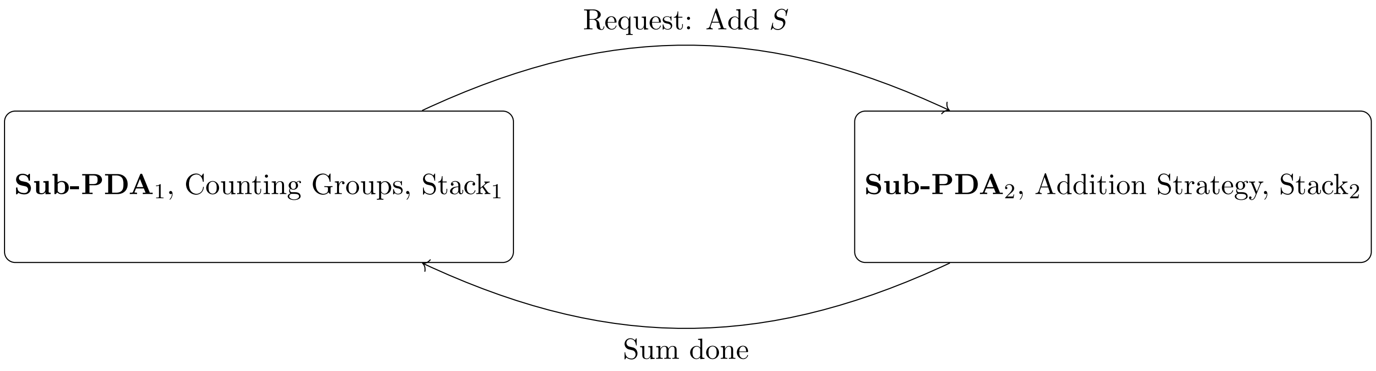

Request: Add S
Sub-PDA$_1$, Counting Groups, Stack$_1$
Sub-PDA$_2$, Addition Strategy, Stack$_2$
Sum done

Figure 2: Two-Stack Composition for Strategic Counting

How it Works:

1. **Initialize**: Sub-PDA$_1$ stores the total number of groups N in Stack$_1$. Sub-PDA$_2$ sets up partial sum $T = 0$ in Stack$_2$.
2. **Repeated Addition**:
   - Sub-PDA$_1$ checks if there is another group left (e.g., by decrementing N).
   - If yes, it triggers Sub-PDA$_2$ to add S to T using a more advanced approach (chunking, rearranging to make bases, etc.).
   - Once Sub-PDA$_2$ completes the addition, control goes back to Sub-PDA$_1$.
3. **Accept**: When Sub-PDA$_1$ has processed all groups, the 2-PDA enters an accepting pair of states with the final sum in Sub-PDA$_2$.

Conclusion

- **High-Level FSA**: Useful for illustrating repeated addition of group sizes without specifying the internal steps of addition.
- **Two-Stack Automaton**: A precise way to compose a simple group-counting sub-PDA and a more sophisticated sub-PDA for strategic addition.

References

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

Strategic Multiplicative Reasoning: Division - Conversion to Groups Other than Bases (CBO)

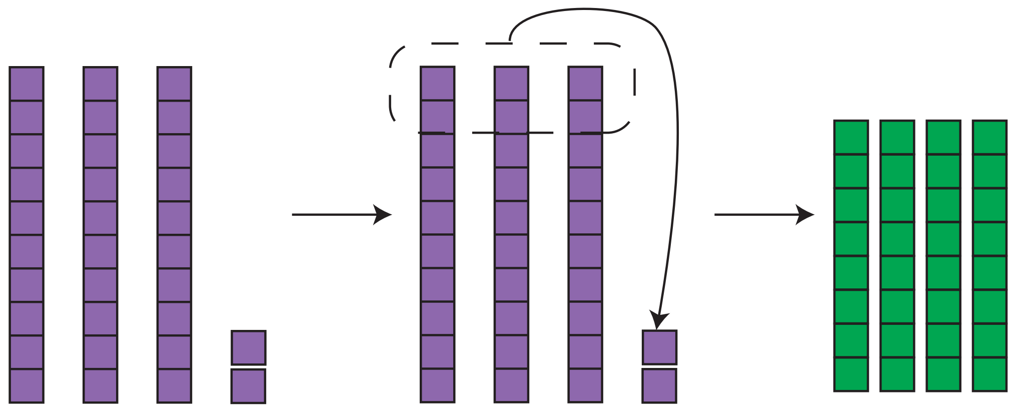
Compiled by: Theodore M. Savich

March 31, 2025

Transcript

Strategy descriptions and examples adapted from Hackenberg (2025).

- **Teacher:**There are 8 pencils in a package. You buy some packages at the store, and have a total of 32 pencils. Determine the total number of packages you bought.
- **Student:** Well, I know that there are 3 groups of ten pencils in 32 pencils. I could make 3 packages of 8 pencils from the 3 groups of ten. Then I would be left with 8 pencils (2 from each ten and 2 more from the units), which would make a fourth package of 8.
- **Teacher:** Great!



$$\begin{aligned} 32 &= 3 \times 10 + 2 \\ &= 3 \times 8 + 3 \times 2 + 2 \\ &= 3 \times 8 + 8 \\ &= 4 \times 8 \\ 32 \div 8 &= 4 \end{aligned}$$

Thus, the total number of packages bought is 4.

Begin with a collection of bases and individual ones—this represents the total number of items. Identify the fixed group size contained within each base. Then, remove an equal number of individual ones from every base to form complete groups of that size, and combine any leftover ones to create additional groups.

For CGOB, using block diagrams works well because they illustrate how an equal number of ones is taken from each base to form groups of the predetermined size, and how those ones can be rearranged to complete the groups.

Conversion to Groups Other than Bases

Strategy Overview

Conversion to Groups Other than Bases involves reorganizing the total number of items into groups that are not aligned with the base system (e.g., base twelve). This strategy is useful when the group size does not neatly fit into the base units, requiring a flexible approach to grouping.

Automaton Design

We design a **Pushdown Automaton (PDA)** that converts a total number of items into groups of a specified size (which is different from the standard base). The PDA uses two stacks: one for tracking the total items and another for forming the new groups.

Components of the PDA

- **States:**
  1. q_{start}: Start state.
  2. q_{read}: Reads the total number of items.
  3. q_{group}: Forms new groups.
  4. q_{output}: Outputs the new grouping.
  5. q_{accept}: Accepting state.
- **Input Alphabet:** $\Sigma = \{E\}$, where E represents an element.
- **Stack Alphabet:** $\Gamma = \{\#, G, E_1, E_2, \ldots\}$, where:
  - $\#$ is the bottom-of-stack marker.
  - G represents a group identifier.
  - E_n represents an element (or the count of elements in a group).
- **Initial Stack Symbol:** $\#$

Automaton Behavior

1. **Initialization:**
   - Begin in q_{start} and push $\#$ onto the stack.
   - Transition to q_{read} to start reading the total number of items.
2. **Reading Total Items:**
   - In q_{read}, for each element E read from the input, push E onto the stack.
   - When all inputs have been read, transition to q_{group}.
3. **Forming New Groups:**

- In q_{group}, pop a fixed number n of E symbols (representing the desired group size) and then push a group identifier G onto the stack.
- Repeat this process until all elements have been grouped.

4. **Outputting New Grouping:**
   - In q_{output}, traverse the stack to read the new grouping.
   - Transition to q_{accept} when the grouping is complete.

Automaton Diagram

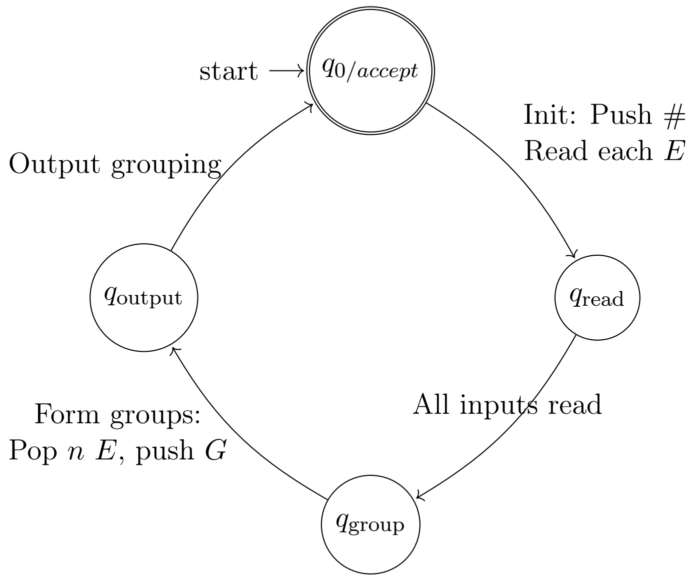


Figure 1: PDA for Conversion to Groups Other than Bases

Example Execution

Problem: Convert 32 items into groups of 8 in base ten.

1. **Initialization:**
   - Start with the stack: #.
2. **Reading Total Items:**
   - Read 32 elements, pushing 32 E symbols onto the stack.
3. **Forming Groups of 8:**
   - Pop 8 E symbols and push G onto the stack.
   - Repeat this process 4 times to form 4 groups.
4. **Final Stack Configuration:** $\# \, G \, G \, G \, G$

Recursive Handling of Group Formation

The PDA recursively forms groups by repeatedly popping a fixed number of elements and pushing a group identifier until all elements are grouped. This ensures the conversion of the total into groups that are not aligned with the standard base system.

HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Division: Conversion to Groups Other than Bases (CGOB)</title>
    <style>
        body { font-family: sans-serif; }
        #cgobDiagram { border: 1px solid #d3d3d3; min-height: 500px; width: 100%; }
        #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;}
        .notation-step { margin-bottom: 0.5em; }
        /* 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-total-block { fill: purple; } /* Color for initial total */
        .final-group-block { fill: lightgreen; } /* Color for final groups */
        .regrouped-block { fill: orange; opacity: 0.7; }
        .remainder-block { fill: lightblue; }
        .regroup-arrow {
            fill: none; stroke: orange; stroke-width: 1.5;
            marker-end: url(#arrowhead-orange);
        }
        .regroup-grouping {
            fill: none; stroke: #555; stroke-width: 1;
            stroke-dasharray: 4 4;
        }
        .section-container {
            margin-bottom: 20px;
            padding: 10px;
            border-radius: 5px;
            background-color: #f9f9f9;
        }
        .final-group {
            stroke: blue;
            stroke-width: 1;
            fill: none;
            stroke-dasharray: 5 3;
        }
    </style>
</head>
<body>

<h1>Strategic Multiplicative Reasoning: Division - Conversion to Groups Other than Bases
    (CGOB)</h1>

<div>
    <label for="cgobTotal">Total Items (Dividend):</label>
    <input type="number" id="cgobTotal" value="32" min="1">
```

```
</div>
<div>
    <label for="cgobGroupSize">Items per Group (Divisor):</label>
    <input type="number" id="cgobGroupSize" value="8" min="1">
</div>

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

<div id="outputContainer">
    <h2>Explanation (Notation):</h2>
    <div id="cgobOutput">
        <!-- Text output will be displayed here -->
    </div>
</div>

<h2>Diagram:</h2>
<div style="overflow-x: auto; overflow-y: auto; max-height: 800px;">
    <svg id="cgobDiagram" preserveAspectRatio="xMinYMin meet">
        <defs>
            <marker id="arrowhead-orange" markerWidth="10" markerHeight="7" refX="9" refY=
                "3.5" orient="auto">
                <polygon points="0 0, 10 3.5, 0 7" fill="orange" />
            </marker>
        </defs>
    </svg>
</div>

<script>
    // --- Helper SVG Functions ---
    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 };
    }

    function drawTenBlock(svg, x, y, width, height, fill, unitBlockSize) {
        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 drawHundredBlock(svg, x, y, size, fill, unitBlockSize) {
        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', size); backgroundRect.setAttribute('height',
            size);
        backgroundRect.setAttribute('fill', fill);
        backgroundRect.setAttribute('class', 'hundred-block-bg␣block');
        group.appendChild(backgroundRect);

        for (let row = 0; row < 10; row++) {
            for (let col = 0; col < 10; col++) {
                const unitBlock = document.createElementNS("http://www.w3.org/2000/svg", '
                    rect');
                unitBlock.setAttribute('x', x + col * unitBlockSize);
                unitBlock.setAttribute('y', y + row * 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: size, height: size, type: 'h', cx: x + size/2, cy: y + size
            /2};
    }

    function createText(svg, x, y, textContent, className = 'diagram-label', anchor = '
        start') {
        const uniqueId = 'text-' + Math.random().toString(36).substr(2, 9);

        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.setAttribute('id', uniqueId);
        text.textContent = textContent;

        if (className === 'diagram-label') {
            const background = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
            const padding = 3;
            const estimatedWidth = Math.max(7 * textContent.length, 30);
            const estimatedHeight = 16;

            let bgX = x - padding;
            if (anchor === 'middle') {
                bgX = x - (estimatedWidth / 2) - padding;
            } else if (anchor === 'end') {
                bgX = x - estimatedWidth - padding;
            }

            background.setAttribute('x', bgX);
            background.setAttribute('y', y - estimatedHeight + padding);
            background.setAttribute('width', estimatedWidth + (padding * 2));
            background.setAttribute('height', estimatedHeight + padding);
            background.setAttribute('fill', 'white');
            background.setAttribute('fill-opacity', '0.9');
            background.setAttribute('rx', '3');
            svg.appendChild(background);
        }

        svg.appendChild(text);
        return uniqueId;
    }

    function createCurvedArrow(svg, x1, y1, x2, y2, cx, cy, arrowClass='regroup-arrow', headId='arrowhead-orange') {
        const path = document.createElementNS("http://www.w3.org/2000/svg", 'path');
        path.setAttribute('d', `M ${x1} ${y1} Q ${cx} ${cy} ${x2} ${y2}`);
        path.setAttribute('class', arrowClass);
        path.setAttribute('marker-end', `url(#${headId})`);
        svg.appendChild(path);
    }

    document.addEventListener('DOMContentLoaded', function() {
        const outputElement = document.getElementById('cgobOutput');
        const totalInput = document.getElementById('cgobTotal');
        const groupSizeInput = document.getElementById('cgobGroupSize');
        const diagramSVG = document.getElementById('cgobDiagram');

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

        window.runCGOBAutomaton = function() {
            try {
                const totalItems = parseInt(totalInput.value);
                const groupSize = parseInt(groupSizeInput.value);

                if (isNaN(totalItems) || isNaN(groupSize) || totalItems <= 0 || groupSize <= 0) {
```

```
                outputElement.textContent = "Please␣enter␣valid␣positive␣numbers";
                diagramSVG.innerHTML = ''; return;
            }

            const numGroups = Math.floor(totalItems / groupSize);
            const remainder = totalItems % groupSize;

            generateCGOBNotation(outputElement, totalItems, groupSize, numGroups, remainder);
            drawCGOBDiagram('cgobDiagram', totalItems, groupSize, numGroups, remainder);

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

    function generateCGOBNotation(outputElement, totalItems, groupSize, numGroups, remainder) {
        let output = `<h2>Conversion to Groups Other than Bases (CGOB) - Notation</h2>`;
        output += `<div class="notation-step"><p class="notation-line␣problem">${totalItems}  ${groupSize} = ?</p></div>`;

        const placeValues = decomposeNumber(totalItems);
        output += `<div class="notation-step"><p class="notation-line">Start with ${totalItems} = ${placeValues.join('␣+␣')}</p></div>`;

        let steps = [];
        let remainders = [];
        let regroupedItems = 0;
        let runningTotal = totalItems;

        let completeGroups = 0;

        for (let i = 0; i < placeValues.length; i++) {
            const placeValue = parseInt(placeValues[i]);
            if (placeValue === 0) continue;

            const base = Math.pow(10, placeValues.length - i - 1);
            const count = placeValue / base;

            if (base > 1) {
                const wholeGroups = Math.floor(base / groupSize);
                const leftover = base % groupSize;

                steps.push(`${count}  ${base} = ${count}  (${wholeGroups}  ${groupSize} + ${leftover})`);
                steps.push(`= ${count * wholeGroups}  ${groupSize} + ${count}  ${leftover}`);

                const newGroups = count * wholeGroups;
                completeGroups += newGroups;
```

```
                    regroupedItems += newGroups * groupSize;

                    if (i > 0 || count * leftover > 0) {
                        steps.push(`= ${completeGroups}  ${groupSize} + ${count * leftover}${remainders.length > 0 ? ' + ' + remainders.join(' + ') : ''} = ${totalItems}`);
                    } else {
                        steps.push(`= ${completeGroups}  ${groupSize} = ${regroupedItems}`);
                    }

                    if (leftover > 0) {
                        remainders.push(count * leftover);
                    }
                } else {
                    remainders.push(placeValue);
                }
            }

            if (remainders.length > 0) {
                const totalRemainder = remainders.reduce((sum, val) => sum + val, 0);
                steps.push(`Combined leftovers: ${remainders.join(' + ')} = ${totalRemainder}`);

                const additionalGroups = Math.floor(totalRemainder / groupSize);
                const finalRemainder = totalRemainder % groupSize;

                if (additionalGroups > 0) {
                    steps.push(`Leftovers form ${additionalGroups} more group${additionalGroups > 1 ? 's' : ''} of ${groupSize}${finalRemainder > 0 ? ` with ${finalRemainder} remaining` : ''}`);

                    completeGroups += additionalGroups;
                    steps.push(`= ${completeGroups}  ${groupSize}${finalRemainder > 0 ? ` + ${finalRemainder}` : ''} = ${totalItems}`);
                } else if (totalRemainder > 0) {
                    steps.push(`= ${completeGroups}  ${groupSize} + ${totalRemainder} = ${totalItems}`);
                }
            }

            steps.forEach(step => {
                output += `<div class="notation-step"><p class="notation-line">${step}</p></div>`;
            });

            output += `<div class="notation-step"><p class="notation-line problem">Result: ${numGroups} groups${remainder > 0 ? ` with ${remainder} remaining` : ''}</p></div>`;
            outputElement.innerHTML = output;
        }

        function decomposeNumber(num) {
            const result = [];
            let tempNum = num;
            let placeValue = Math.pow(10, Math.floor(Math.log10(num)));

            while (placeValue >= 1) {
                const digit = Math.floor(tempNum / placeValue);
                if (digit > 0) {
                    result.push(digit * placeValue);
                }
                tempNum %= placeValue;
                placeValue /= 10;
            }

            return result;
        }

        function drawCGOBDiagram(svgId, totalItems, groupSize, numGroups, remainder) {
            const svg = document.getElementById(svgId);
            if (!svg) return;

            svg.innerHTML = '';

            const defs = document.createElementNS("http://www.w3.org/2000/svg", 'defs');
            defs.innerHTML = `<marker id="arrowhead-orange" markerWidth="10" markerHeight=
                "7"
                             refX="9" refY="3.5" orient="auto">
                             <polygon points="0 0, 10 3.5, 0 7" fill="orange" /></marker>`;
            svg.appendChild(defs);

            const blockUnitSize = totalItems > 100 ? 8 : 10;
            const tenBlockWidth = blockUnitSize;
            const tenBlockHeight = blockUnitSize * 10;
            const hundredBlockSize = blockUnitSize * 10;
            const blockSpacing = 4;
            const groupSpacingX = 20;
            const sectionSpacingY = 150;
            const startX = 30;
            let currentY = 40;

            const colorInitial = 'purple';
            const colorFinal = 'lightgreen';
            const colorRemainder = 'lightblue';
            const colorRegrouped = 'orange';

            const maxBlockHeight = Math.max(tenBlockHeight, hundredBlockSize,
                blockUnitSize);

            createText(svg, startX, currentY, `Initial Total: ${totalItems}`);
            currentY += 30;
            let currentX = startX;
            let section1MaxY = currentY;

            let hundreds = Math.floor(totalItems / 100);
            let tens = Math.floor((totalItems % 100) / 10);
            let ones = totalItems % 10;
```

```

            let initialBlocksData = [];

            for (let i = 0; i < hundreds; i++) {
                let info = drawHundredBlock(svg, currentX, currentY, hundredBlockSize, colorInitial, blockUnitSize);
                initialBlocksData.push(info);
                currentX += hundredBlockSize + groupSpacingX;
                section1MaxY = Math.max(section1MaxY, currentY + hundredBlockSize);
            }

            for (let i = 0; i < tens; i++) {
                let info = drawTenBlock(svg, currentX, currentY, tenBlockWidth, tenBlockHeight, colorInitial, blockUnitSize);
                initialBlocksData.push(info);
                currentX += tenBlockWidth + blockSpacing;
                section1MaxY = Math.max(section1MaxY, currentY + tenBlockHeight);
            }

            for (let i = 0; i < ones; i++) {
                let info = drawBlock(svg, currentX, currentY + maxBlockHeight - blockUnitSize, blockUnitSize, colorInitial);
                initialBlocksData.push(info);
                currentX += blockUnitSize + blockSpacing;
                section1MaxY = Math.max(section1MaxY, currentY + blockUnitSize);
            }

            currentY = section1MaxY + sectionSpacingY;

            createText(svg, startX, currentY, ‘Regrouping into groups of ${groupSize}‘);
            currentY += 30;

            let allRegroupData = visualizeRegrouping(svg, startX, currentY, totalItems, groupSize,
                                                    blockUnitSize, blockSpacing, groupSpacingX,
                                                    colorRegrouped, colorRemainder);

            let section2MaxY = allRegroupData.maxY;

            currentY = section2MaxY + sectionSpacingY;

            createText(svg, startX, currentY, ‘Final Result: ${numGroups} groups of ${groupSize}${remainder > 0 ? ‘ with ${remainder} remaining‘ : ’’}‘);
            currentY += 30;

            let section3MaxY = drawFinalGroups(svg, startX, currentY, numGroups, groupSize, remainder,
                                              blockUnitSize, blockSpacing, groupSpacingX, colorFinal, colorRemainder);

            if (allRegroupData.groups && allRegroupData.groups.length > 0) {
                drawConnectionArrows(svg, allRegroupData.groups, startX, currentY,
                                     numGroups, groupSize, blockUnitSize, blockSpacing, groupSpacingX);
            }

            let itemsPerRow = Math.min(groupSize, 8);
            let finalGroupWidth = (itemsPerRow * (blockUnitSize + blockSpacing)) - blockSpacing + 8;
            let finalGroupHeight = (Math.ceil(groupSize / itemsPerRow) * (blockUnitSize + blockSpacing)) - blockSpacing + 8;
            let labelOffset = Math.min(25, finalGroupHeight / 3);

            let svgWidth = Math.max(800, currentX + 100);
            let svgHeight = Math.max(section3MaxY + 50, currentY + finalGroupHeight);

            const maxGroupsPerRow = Math.floor((650 - 50) / (finalGroupWidth + groupSpacingX));
            const numFinalRows = Math.ceil(numGroups / maxGroupsPerRow);
            if (numFinalRows > 1) {
                svgHeight += (numFinalRows - 1) * (finalGroupHeight + groupSpacingX + labelOffset);
            }

            svg.setAttribute('width', svgWidth);
            svg.setAttribute('height', svgHeight);
            svg.setAttribute('viewBox', `0 0 ${svgWidth} ${svgHeight}`);
        }

        function visualizeRegrouping(svg, startX, startY, totalItems, groupSize,
                                     blockSize, blockSpacing, groupSpacing,
                                     regroupColor, remainderColor) {
            let currentX = startX;
            let currentY = startY;
            let maxY = currentY;

            let hundreds = Math.floor(totalItems / 100);
            let tens = Math.floor((totalItems % 100) / 10);
            let ones = totalItems % 10;

            let allRegroupedGroups = [];
            let allLeftovers = [];

            for (let h = 0; h < hundreds; h++) {
                let groupsPerHundred = Math.floor(100 / groupSize);
                let leftoverPerHundred = 100 % groupSize;

                const hundredSize = 10 * blockSize;

                const hundredOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                hundredOutline.setAttribute('x', currentX);
                hundredOutline.setAttribute('y', currentY);
                hundredOutline.setAttribute('width', hundredSize);
                hundredOutline.setAttribute('height', hundredSize);
                hundredOutline.setAttribute('fill', 'none');
```

```
                hundredOutline.setAttribute('stroke', 'gray');
                hundredOutline.setAttribute('stroke-dasharray', '4 4');
                svg.appendChild(hundredOutline);

                const unitsPerRow = 10;
                const unitsPerCol = 10;
                const fullRows = Math.floor(groupSize / unitsPerRow);
                const remainingInLastRow = groupSize % unitsPerRow;

                for (let g = 0; g < groupsPerHundred; g++) {
                    let startRow = Math.floor((g * groupSize) / unitsPerRow);
                    let startCol = (g * groupSize) % unitsPerRow;

                    for (let i = 0; i < groupSize; i++) {
                        let row = startRow + Math.floor((startCol + i) / unitsPerRow);
                        let col = (startCol + i) % unitsPerRow;

                        if (row < unitsPerCol) {
                            const unitRect = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                            unitRect.setAttribute('x', currentX + col * blockSize);
                            unitRect.setAttribute('y', currentY + row * blockSize);
                            unitRect.setAttribute('width', blockSize);
                            unitRect.setAttribute('height', blockSize);
                            unitRect.setAttribute('fill', regroupColor);
                            unitRect.setAttribute('opacity', '0.7');
                            svg.appendChild(unitRect);
                        }
                    }

                    let groupStartRow = Math.floor((g * groupSize) / unitsPerRow);
                    let groupStartCol = (g * groupSize) % unitsPerRow;
                    let groupEndRow = Math.floor(((g+1) * groupSize - 1) / unitsPerRow);
                    let groupEndCol = ((g+1) * groupSize - 1) % unitsPerRow;

                    if (groupStartRow === groupEndRow) {
                        const groupOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                        groupOutline.setAttribute('x', currentX + groupStartCol * blockSize - 1);
                        groupOutline.setAttribute('y', currentY + groupStartRow * blockSize - 1);
                        groupOutline.setAttribute('width', (groupEndCol - groupStartCol + 1) * blockSize + 2);
                        groupOutline.setAttribute('height', blockSize + 2);
                        groupOutline.setAttribute('fill', 'none');
                        groupOutline.setAttribute('stroke', '#555');
                        groupOutline.setAttribute('stroke-dasharray', '4 4');
                        svg.appendChild(groupOutline);

                        allRegroupedGroups.push({
                            x: currentX + groupStartCol * blockSize,
                            y: currentY + groupStartRow * blockSize,
                            width: (groupEndCol - groupStartCol + 1) * blockSize,
                            height: blockSize,
                            cx: currentX + (groupStartCol + (groupEndCol - groupStartCol)/2) * blockSize,
                            cy: currentY + (groupStartRow + 0.5) * blockSize,
                            isBaseGroup: true
                        });
                    } else {
                        const firstRowWidth = (unitsPerRow - groupStartCol) * blockSize;
                        const firstRowOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                        firstRowOutline.setAttribute('x', currentX + groupStartCol * blockSize - 1);
                        firstRowOutline.setAttribute('y', currentY + groupStartRow * blockSize - 1);
                        firstRowOutline.setAttribute('width', firstRowWidth + 2);
                        firstRowOutline.setAttribute('height', blockSize + 2);
                        firstRowOutline.setAttribute('fill', 'none');
                        firstRowOutline.setAttribute('stroke', '#555');
                        firstRowOutline.setAttribute('stroke-dasharray', '4␣4');
                        svg.appendChild(firstRowOutline);

                        for (let r = groupStartRow + 1; r < groupEndRow; r++) {
                            const rowOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                            rowOutline.setAttribute('x', currentX - 1);
                            rowOutline.setAttribute('y', currentY + r * blockSize - 1);
                            rowOutline.setAttribute('width', unitsPerRow * blockSize + 2);
                            rowOutline.setAttribute('height', blockSize + 2);
                            rowOutline.setAttribute('fill', 'none');
                            rowOutline.setAttribute('stroke', '#555');
                            rowOutline.setAttribute('stroke-dasharray', '4␣4');
                            svg.appendChild(rowOutline);
                        }

                        const lastRowWidth = (groupEndCol + 1) * blockSize;
                        const lastRowOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                        lastRowOutline.setAttribute('x', currentX - 1);
                        lastRowOutline.setAttribute('y', currentY + groupEndRow * blockSize - 1);
                        lastRowOutline.setAttribute('width', lastRowWidth + 2);
                        lastRowOutline.setAttribute('height', blockSize + 2);
                        lastRowOutline.setAttribute('fill', 'none');
                        lastRowOutline.setAttribute('stroke', '#555');
                        lastRowOutline.setAttribute('stroke-dasharray', '4␣4');
                        svg.appendChild(lastRowOutline);

                        allRegroupedGroups.push({
                            x: currentX,
                            y: currentY + groupStartRow * blockSize,
                            width: hundredSize,
                            height: (groupEndRow - groupStartRow + 1) * blockSize,
                            cx: currentX + hundredSize/2,
                            cy: currentY + (groupStartRow + (groupEndRow - groupStartRow)/2) * blockSize,
                            isBaseGroup: true
                        });
                    }
                }

                if (leftoverPerHundred > 0) {
                    let leftoverStartRow = Math.floor((groupsPerHundred * groupSize) / unitsPerRow);
                    let leftoverStartCol = (groupsPerHundred * groupSize) % unitsPerRow;

                    for (let i = 0; i < leftoverPerHundred; i++) {
                        let row = leftoverStartRow + Math.floor((leftoverStartCol + i) / unitsPerRow);
                        let col = (leftoverStartCol + i) % unitsPerRow;

                        if (row < unitsPerCol) {
                            const unitRect = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                            unitRect.setAttribute('x', currentX + col * blockSize);
                            unitRect.setAttribute('y', currentY + row * blockSize);
                            unitRect.setAttribute('width', blockSize);
                            unitRect.setAttribute('height', blockSize);
                            unitRect.setAttribute('fill', remainderColor);
                            svg.appendChild(unitRect);
                        }
                    }

                    let leftoverEndRow = Math.floor(((groupsPerHundred * groupSize) + leftoverPerHundred - 1) / unitsPerRow);
                    let leftoverEndCol = ((groupsPerHundred * groupSize) + leftoverPerHundred - 1) % unitsPerRow;

                    if (leftoverStartRow === leftoverEndRow) {
                        const leftoverOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                        leftoverOutline.setAttribute('x', currentX + leftoverStartCol * blockSize - 1);
                        leftoverOutline.setAttribute('y', currentY + leftoverStartRow * blockSize - 1);
                        leftoverOutline.setAttribute('width', (leftoverEndCol - leftoverStartCol + 1) * blockSize + 2);
                        leftoverOutline.setAttribute('height', blockSize + 2);
                        leftoverOutline.setAttribute('fill', 'none');
                        leftoverOutline.setAttribute('stroke', '#555');
                        leftoverOutline.setAttribute('stroke-dasharray', '4 4');
                        svg.appendChild(leftoverOutline);

                        allLeftovers.push({
                            count: leftoverPerHundred,
                            info: {
                                x: currentX + leftoverStartCol * blockSize,
                                y: currentY + leftoverStartRow * blockSize,
                                width: (leftoverEndCol - leftoverStartCol + 1) * blockSize,
                                height: blockSize,
                                cx: currentX + (leftoverStartCol + (leftoverEndCol -
                                    leftoverStartCol)/2) * blockSize,
                                cy: currentY + (leftoverStartRow + 0.5) * blockSize
                            }
                        });
                    } else {
                        allLeftovers.push({
                            count: leftoverPerHundred,
                            info: {
                                x: currentX,
                                y: currentY + leftoverStartRow * blockSize,
                                width: hundredSize,
                                height: (leftoverEndRow - leftoverStartRow + 1) * blockSize,
                                cx: currentX + hundredSize/2,
                                cy: currentY + (leftoverStartRow + (leftoverEndRow -
                                    leftoverStartRow)/2) * blockSize
                            }
                        });
                    }
                }

                currentX += hundredSize + groupSpacing;
                maxY = Math.max(maxY, currentY + hundredSize);
            }

            for (let t = 0; t < tens; t++) {
                let groupsPerTen = Math.floor(10 / groupSize);
                let leftoverPerTen = 10 % groupSize;

                const tenHeight = 10 * blockSize;
                const tenWidth = blockSize;

                const tenOutline = document.createElementNS("http://www.w3.org/2000/svg", '
                    rect');
                tenOutline.setAttribute('x', currentX);
                tenOutline.setAttribute('y', currentY);
                tenOutline.setAttribute('width', tenWidth);
                tenOutline.setAttribute('height', tenHeight);
                tenOutline.setAttribute('fill', 'none');
                tenOutline.setAttribute('stroke', 'gray');
                tenOutline.setAttribute('stroke-dasharray', '4␣4');
                svg.appendChild(tenOutline);

                if (groupsPerTen > 0) {
                    for (let g = 0; g < groupsPerTen; g++) {
                        const groupRect = document.createElementNS("http://www.w3.org/2000/
                            svg", 'rect');
                        groupRect.setAttribute('x', currentX);
                        groupRect.setAttribute('y', currentY + g * groupSize * blockSize);
                        groupRect.setAttribute('width', tenWidth);
                        groupRect.setAttribute('height', groupSize * blockSize);
                        groupRect.setAttribute('fill', regroupColor);
```

```
                        groupRect.setAttribute('opacity', '0.7');
                        svg.appendChild(groupRect);

                        const groupOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                        groupOutline.setAttribute('x', currentX - 1);
                        groupOutline.setAttribute('y', currentY + g * groupSize * blockSize - 1);
                        groupOutline.setAttribute('width', tenWidth + 2);
                        groupOutline.setAttribute('height', groupSize * blockSize + 2);
                        groupOutline.setAttribute('fill', 'none');
                        groupOutline.setAttribute('stroke', '#555');
                        groupOutline.setAttribute('stroke-dasharray', '4␣4');
                        svg.appendChild(groupOutline);

                        allRegroupedGroups.push({
                            x: currentX,
                            y: currentY + g * groupSize * blockSize,
                            width: tenWidth,
                            height: groupSize * blockSize,
                            cx: currentX + tenWidth/2,
                            cy: currentY + (g * groupSize + groupSize/2) * blockSize,
                            isBaseGroup: true
                        });
                    }
                }

                if (leftoverPerTen > 0) {
                    const leftoverY = currentY + groupsPerTen * groupSize * blockSize;

                    const leftoverRect = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                    leftoverRect.setAttribute('x', currentX);
                    leftoverRect.setAttribute('y', leftoverY);
                    leftoverRect.setAttribute('width', tenWidth);
                    leftoverRect.setAttribute('height', leftoverPerTen * blockSize);
                    leftoverRect.setAttribute('fill', remainderColor);
                    svg.appendChild(leftoverRect);

                    const leftoverOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                    leftoverOutline.setAttribute('x', currentX - 1);
                    leftoverOutline.setAttribute('y', leftoverY - 1);
                    leftoverOutline.setAttribute('width', tenWidth + 2);
                    leftoverOutline.setAttribute('height', leftoverPerTen * blockSize + 2);
                    leftoverOutline.setAttribute('fill', 'none');
                    leftoverOutline.setAttribute('stroke', '#555');
                    leftoverOutline.setAttribute('stroke-dasharray', '4␣4');
                    svg.appendChild(leftoverOutline);

                    allLeftovers.push({
                        count: leftoverPerTen,
                        info: {
                            x: currentX,
                            y: leftoverY,
                            width: tenWidth,
                            height: leftoverPerTen * blockSize,
                            cx: currentX + tenWidth/2,
                            cy: leftoverY + leftoverPerTen * blockSize/2
                        }
                    });
                }

                currentX += tenWidth + blockSpacing;
                maxY = Math.max(maxY, currentY + tenHeight);
            }

            if (ones > 0) {
                let onesStartX = currentX;

                for (let i = 0; i < ones; i++) {
                    let oneBlock = drawBlock(svg, currentX, currentY, blockSize, remainderColor);

                    allLeftovers.push({
                        count: 1,
                        info: oneBlock
                    });

                    currentX += blockSize + blockSpacing;
                }

                if (ones > 1) {
                    const onesOutline = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
                    onesOutline.setAttribute('x', onesStartX - 1);
                    onesOutline.setAttribute('y', currentY - 1);
                    onesOutline.setAttribute('width', ones * (blockSize + blockSpacing) - blockSpacing + 2);
                    onesOutline.setAttribute('height', blockSize + 2);
                    onesOutline.setAttribute('fill', 'none');
                    onesOutline.setAttribute('stroke', '#555');
                    onesOutline.setAttribute('stroke-dasharray', '4 4');
                    svg.appendChild(onesOutline);
                }

                maxY = Math.max(maxY, currentY + blockSize);
            }

            if (allLeftovers.length > 0) {
                let totalLeftover = allLeftovers.reduce((sum, item) => sum + item.count, 0);

                if (totalLeftover >= groupSize) {
                    let additionalGroups = Math.floor(totalLeftover / groupSize);
                    let finalRemainder = totalLeftover % groupSize;

                    currentY = maxY + 40;
                    createText(svg, startX, currentY, `Combined Leftovers: ${totalLeftover} items`);
                    currentY += 30;
                    currentX = startX;

                    let combinedGroupsInfo = [];

                    for (let g = 0; g < additionalGroups; g++) {
                        let groupInfo = drawRegroupBlock(svg, currentX, currentY, groupSize, blockSize,
                                                      blockSpacing, regroupColor, true);

                        createText(svg, currentX + groupInfo.width/2, currentY - 15,
                                 `Group ${g+1} from Leftovers`, 'diagram-label', 'middle');

                        combinedGroupsInfo.push(groupInfo);
                        currentX += groupInfo.width + groupSpacing * 1.5;
                    }

                    if (finalRemainder > 0) {
                        let remainderInfo = drawRegroupBlock(svg, currentX, currentY, finalRemainder,
                                                          blockSize, blockSpacing, remainderColor, false);

                        createText(svg, currentX + remainderInfo.width/2, currentY - 15,
                                 `Final Remainder`, 'diagram-label', 'middle');

                        maxY = Math.max(maxY, currentY + remainderInfo.height);
                    } else {
                        maxY = Math.max(maxY, currentY + (Math.ceil(groupSize/8) * (blockSize + blockSpacing)));
                    }

                    let targetX = startX + (additionalGroups * groupSize * blockSize / 4);
                    let targetY = currentY - 25;

                    for (let leftover of allLeftovers) {
                        let source = leftover.info;

                        createCurvedArrow(svg, source.cx, source.cy,
                                        targetX, targetY,
                                        (source.cx + targetX)/2, (source.cy + targetY)/2 - 20);
                    }

                    allRegroupedGroups.push(...combinedGroupsInfo);
                }
            }

            return {
                maxY: maxY,
                groups: allRegroupedGroups,
                leftovers: allLeftovers
            };
        }

        function drawRegroupBlock(svg, x, y, count, blockSize, blockSpacing, color,
            addOutline = true) {
            let itemsPerRow = Math.min(count, 8);
            let rows = Math.ceil(count / itemsPerRow);

            let groupWidth = (itemsPerRow * (blockSize + blockSpacing)) - blockSpacing;
            let groupHeight = (rows * (blockSize + blockSpacing)) - blockSpacing;

            for (let i = 0; i < count; i++) {
                let row = Math.floor(i / itemsPerRow);
                let col = i % itemsPerRow;

                drawBlock(svg,
                        x + col * (blockSize + blockSpacing),
                        y + row * (blockSize + blockSpacing),
                        blockSize, color);
            }

            if (addOutline) {
                const outline = document.createElementNS("http://www.w3.org/2000/svg", '
                    rect');
                outline.setAttribute('x', x - 2);
                outline.setAttribute('y', y - 2);
                outline.setAttribute('width', groupWidth + 4);
                outline.setAttribute('height', groupHeight + 4);
                outline.setAttribute('fill', 'none');
                outline.setAttribute('stroke', '#555');
                outline.setAttribute('stroke-dasharray', '4␣4');
                svg.appendChild(outline);
            }

            return {
                x: x,
                y: y,
                width: groupWidth,
                height: groupHeight,
                cx: x + groupWidth/2,
                cy: y + groupHeight/2
            };
        }

        function drawConnectionArrows(svg, sourceGroups, targetStartX, targetStartY,
                                 numGroups, groupSize, blockSize, blockSpacing,
                                        groupSpacing) {
            const combinedGroups = sourceGroups.filter(group => !group.isBaseGroup);
            if (combinedGroups.length === 0) return;

            const baseGroupCount = sourceGroups.filter(group => group.isBaseGroup).length;

            let itemsPerRow = Math.min(groupSize, 8);
            let groupWidth = (itemsPerRow * (blockSize + blockSpacing)) - blockSpacing + 8;
            let groupHeight = (Math.ceil(groupSize / itemsPerRow) * (blockSize + blockSpacing)) - blockSpacing + 8;
            let labelOffset = Math.min(25, groupHeight / 3);

            const svgContainerWidth = 650;
            const maxGroupsPerRow = Math.max(1, Math.floor((svgContainerWidth - 50) / (groupWidth + groupSpacing)));

            for (let i = 0; i < combinedGroups.length; i++) {
                let source = combinedGroups[i];

                const targetGroupIndex = baseGroupCount + i;

                const targetRow = Math.floor(targetGroupIndex / maxGroupsPerRow);
                const targetCol = targetGroupIndex % maxGroupsPerRow;

                let targetX = targetStartX + (targetCol * (groupWidth + groupSpacing)) + groupWidth/2;
                let arrowEndY = targetStartY + (targetRow * (groupHeight + groupSpacing + labelOffset));

                let controlY = (source.cy + arrowEndY)/2;
                if (arrowEndY < source.cy) {
                    controlY = arrowEndY + (source.cy - arrowEndY)/2;
                }

                createCurvedArrow(
                    svg,
                    source.cx, source.cy + source.height/2 + 5,
                    targetX, arrowEndY + 10,
                    source.cx, controlY
                );
            }
        }

        function drawFinalGroups(svg, startX, startY, numGroups, groupSize, remainder,
                                blockSize, blockSpacing, groupSpacing, groupColor, remainderColor) {
            let maxY = startY;

            let itemsPerRow = Math.min(groupSize, 8);
            let groupWidth = (itemsPerRow * (blockSize + blockSpacing)) - blockSpacing + 8;
            let groupHeight = (Math.ceil(groupSize / itemsPerRow) * (blockSize + blockSpacing)) - blockSpacing + 8;

            const svgContainerWidth = 650;
            const maxGroupsPerRow = Math.max(1, Math.floor((svgContainerWidth - 60) / (groupWidth + groupSpacing)));
            const labelOffset = Math.min(25, groupHeight / 3);

            const numRows = Math.ceil(numGroups / maxGroupsPerRow);
```

```

            for (let row = 0; row < numRows; row++) {
                let currentY = startY + row * (groupHeight + groupSpacing + labelOffset);
                let currentX = startX;

                const startGroup = row * maxGroupsPerRow;
                const endGroup = Math.min(numGroups, (row + 1) * maxGroupsPerRow);

                for (let g = startGroup; g < endGroup; g++) {
                    let groupStartX = currentX;

                    const groupRect = document.createElementNS("http://www.w3.org/2000/svg"
                        , 'rect');
                    groupRect.setAttribute('x', groupStartX - 4);
                    groupRect.setAttribute('y', currentY - 4);
                    groupRect.setAttribute('width', groupWidth);
                    groupRect.setAttribute('height', groupHeight);
                    groupRect.setAttribute('class', 'final-group');
                    svg.appendChild(groupRect);

                    createText(svg, groupStartX + groupWidth/2, currentY - labelOffset/2,
                            `Group ${g+1}`, 'diagram-label', 'middle');

                    for (let r = 0; r < Math.ceil(groupSize / itemsPerRow); r++) {
                        for (let c = 0; c < itemsPerRow; c++) {
                            const index = r * itemsPerRow + c;
                            if (index < groupSize) {
                                drawBlock(svg,
                                          groupStartX + c * (blockSize + blockSpacing),
                                          currentY + r * (blockSize + blockSpacing),
                                          blockSize, groupColor, 'final-group-block');
                            }
                        }
                    }

                    currentX += groupWidth + groupSpacing;
                    maxY = Math.max(maxY, currentY + groupHeight);
                }
            }

            if (remainder > 0) {
                let remainderY, remainderX;

                if (numRows === 1 && numGroups * (groupWidth + groupSpacing) + remainder *
                     (blockSize + blockSpacing) < svgContainerWidth - 60) {
                    remainderY = startY;
                    remainderX = startX + numGroups * (groupWidth + groupSpacing);
                } else {
                    const lastRowGroups = numGroups % maxGroupsPerRow || maxGroupsPerRow;
                    const remainderWidth = remainder * (blockSize + blockSpacing);

                    if (lastRowGroups * (groupWidth + groupSpacing) + remainderWidth <
                        svgContainerWidth - 60) {
```

```
                        remainderY = startY + (numRows - 1) * (groupHeight + groupSpacing +
                             labelOffset);
                        remainderX = startX + lastRowGroups * (groupWidth + groupSpacing);
                    } else {
                        remainderY = startY + numRows * (groupHeight + groupSpacing +
                            labelOffset);
                        remainderX = startX;
                    }
                }

                createText(svg, remainderX + (remainder * (blockSize + blockSpacing))/2,
                    remainderY - labelOffset/2,
                         `Remainder: ${remainder}`, 'diagram-label', 'middle');

                for (let r = 0; r < remainder; r++) {
                    drawBlock(svg,
                             remainderX + r * (blockSize + blockSpacing),
                             remainderY,
                             blockSize, remainderColor, 'remainder-block');
                }

                maxY = Math.max(maxY, remainderY + blockSize);
            }

            return maxY;
        }

        runCGOBAutomaton();
    });
</script>

</body>
</html>
```

References

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

Division Strategies - Dealing by Ones

Compiled by: Theodore M. Savich

March 31, 2025

This is a sharing division problem. With sharing division problems, the number of items in each group is unknown, while the number of groups and the total number of items are both known.

$\boxed{\text{Number of groups}} \times \boxed{\text{Unknown Number of items in each group}} = \boxed{\text{Total number of items}}$

Transcript

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

- **Teacher:** Mr. Gomez has 12 cupcakes. He wants to put the cupcakes into four boxes, so that there's the same number in each box. How many cupcakes can go in each box?
- **Student:** Okay, 1, 2, 3, 4. I got four boxes, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12. Now, one will go in this box, one will go in this box, one will go in this box, one will go in this box. Two will go in this box, two will go in this box, two will go in this box, two will go in this box. Three will go in this box, three will go in this box, three will go in this box, and three, will go in this box. Three cupcakes can go in each box.
- **Teacher:** Nice. Thank you, Alex.

Alex began by placing 4 unifix cubes of the same color on the table, each one standing in for a different box. He then selected 12 additional cubes to represent 12 cupcakes. One by one, he distributed a cube from this pile to each box, repeating the process until he had used all the cupcake cubes. When he finished, he observed that each box contained 3 cubes, so the answer is 3 cupcakes per box.

Dealing by Ones

Strategy Overview

Dealing by Ones is a foundational division strategy where the division is performed by incrementally removing one item at a time and counting the number of groups formed. This method is particularly useful for simple division problems and serves as the basis for more advanced strategies.

Automaton Design

We design a **Pushdown Automaton (PDA)** that systematically removes one element from the total and increments the group count until all elements have been distributed.

Automaton Tuple

The PDA is defined as the 7-tuple

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

where:

- $Q = \{q_{0/accept}, q_{\text{remove}}, q_{\text{output}}\}$ is the set of states. Here, $q_{0/accept}$ is the merged start and accepting state.
- $\Sigma = \{E\}$ is the input alphabet, where E represents an element.
- $\Gamma = \{\#, G, E\}$ is the stack alphabet:
  - $\#$ is the bottom-of-stack marker.
  - G represents a group identifier.
  - E represents an element.
- $q_{0/accept}$ is the start (and accepting) state.
- $\#$ is the initial stack symbol.
- $F = \{q_{0/accept}\}$ is the set of accepting states.

Transition Function

The key transitions of the PDA are as follows:

1. **Initialization:**
$$\delta(q_{0/accept},\, \varepsilon,\, \varepsilon) = (q_{\text{remove}},\, \#)$$
(Push the bottom marker # and move to the removal phase.)

2. **Removing Elements:**
$$\delta(q_{\text{remove}},\, \varepsilon,\, E) = (q_{\text{remove}},\, \varepsilon \text{ (pop } E\text{) followed by pushing } G)$$
(For each E encountered on the stack, pop it and push G to record one completed group.)

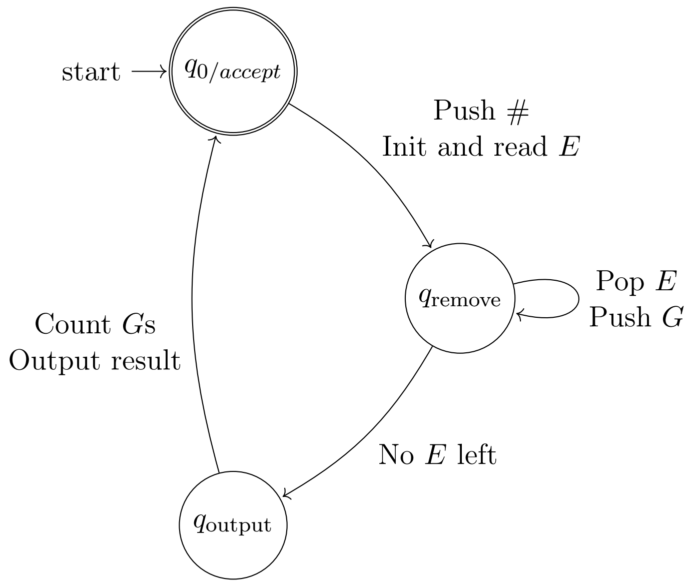
3. **Transition when no E remains:**
$$\delta(q_{\text{remove}},\, \varepsilon,\, \#) = (q_{\text{output}},\, \#)$$
(When no E is left (only the bottom marker remains), move to the output phase.)

4. **Outputting the Result:**
$$\delta(q_{\text{output}},\, \varepsilon,\, x) = (q_{0/accept},\, x)$$
(Count the number of G symbols to determine the quotient; output the final count and return to the merged start/accept state.)

State Transition Table

Current State	**Input Symbol**	**Stack Top**	**Next State**	**Stack Operation**	**Description**
$q_{0/accept}$	ε	—	q_{remove}	Push #	Initialization
q_{remove}	ε	E	q_{remove}	Pop E, push G	Remove one element, increment group count
q_{remove}	ε	#	q_{output}	No change	All E's removed
q_{output}	ε	(Any)	$q_{0/accept}$	Output final count	Output quotient (number of G's)

Circular PDA Diagram



Example Execution

Problem: Divide 7 items into groups of 1.

1. **Start:**
   - Initial Stack: $\# E E E E E E E$ (7 E's representing 7 items).
2. **Removing Elements:**
   - For each E popped, a G is pushed. After 7 removals, the stack becomes: $\# G G G G G G G$.
3. **Outputting the Result:**
   - The automaton counts the 7 G's and outputs the result (7 groups of 1).

HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Division: Dealing by Ones</title>
    <style>
        body { font-family: sans-serif; line-height: 1.6; }
        .container { max-width: 800px; margin: 10px auto; padding: 10px;}
        .control-section, .pile-section, .groups-section, .result-section {
            margin-bottom: 20px; padding: 10px; border: 1px solid #eee;
            background-color: #f9f9f9; border-radius: 5px;
        }
        label { margin-right: 5px;}
        input[type=number] { width: 60px; margin-right: 15px;}
        button { padding: 5px 10px; font-size: 1em; margin-right: 5px; }
        #statusMessage { color: #e65c00; font-weight: bold; margin-left: 15px;}
        .pile-container, .groups-container {
            min-height: 40px; padding: 5px; background-color: #fff; border: 1px dashed #
                ccc;
            margin-top: 5px;
         }
        .group-box {
            display: inline-block; /* Changed from flex */
            vertical-align: top; /* Align boxes at the top */
            width: 100px; /* Fixed width for each group box */
            min-height: 80px;
            border: 1px solid #999;
            padding: 5px;
            margin: 5px;
            background-color: #e8f4ff;
            text-align: center;
         }
         .group-box-label {
           font-size: 0.9em;
           color: #555;
           margin-bottom: 5px;
           display: block;
           min-height: 1.2em; /* Ensure space even if empty */
         }
        .item-block { /* Renamed from .box for clarity */
           display: inline-block;
           width: 12px; height: 12px; margin: 1px;
           background-color: dodgerblue; border: 1px solid #666;
           vertical-align: middle;
        }
         #resultValue { font-size: 1.5em; font-weight: bold; color: darkgreen; }

    </style>
</head>
<body>
<div class="container">

    <h1>Division Strategies - Dealing by Ones</h1>

    <div class="control-section">
        <label for="dealTotalInput">Total Items:</label>
        <input type="number" id="dealTotalInput" value="12" min="0">
        <label for="dealGroupsInput">Number of Groups:</label>
        <input type="number" id="dealGroupsInput" value="4" min="1">
        <!-- Ensure onclick calls the globally exposed functions -->
        <button onclick="setupSimulation()">Set Up / Reset</button>
        <button onclick="dealOneItem()" id="dealBtn" disabled>Deal One Item</button>
         <span id="statusMessage"></span>
    </div>

    <div class="pile-section">
        <strong>Items Remaining in Pile:</strong> <span id="pileCount">0</span>
        <div id="pileDisplay" class="pile-container"></div>
    </div>

    <div class="groups-section">
         <strong>Groups (Dealing items into these):</strong>
         <div id="groupsDisplay" class="groups-container">
             <!-- Group boxes will be added here -->
         </div>
    </div>

     <div class="result-section">
        <strong>Result (Items per group):</strong> <span id="resultValue">?</span>
    </div>


    <script>
        // --- Simulation State Variables (Global in this simple example) ---
        let initialTotalItems = 0;
        let numGroups = 0;
        let itemsRemaining = 0;
        let groupsData = []; // Stores item count for each group: [3, 3, 3, 3]
        let nextGroupIndex = 0;
        let isDealingComplete = true;

        // --- DOM Element References (Get them once DOM is loaded) ---
        let numericValueSpan, resultValueSpan, pileDisplay, pileCountSpan, groupsDisplay,
            dealBtn, statusMessage, totalInput, groupsInput;

        // --- Simulation Functions ---
        // Note: These are defined globally OR attached to window inside DOMContentLoaded

        function updatePileDisplay() {
            if (!pileDisplay || !pileCountSpan) return; // Check if elements exist
            pileCountSpan.textContent = itemsRemaining;
            pileDisplay.innerHTML = ""; // Clear previous
            for (let i = 0; i < itemsRemaining; i++) {
                const item = document.createElement("div");
                item.className = "item-block";
                pileDisplay.appendChild(item);
            }
        }

        function drawGroupContainers() {
            if (!groupsDisplay) return; // Check if element exists
            groupsDisplay.innerHTML = ""; // Clear previous
            for (let i = 0; i < numGroups; i++) {
                const groupBox = document.createElement("div");
                groupBox.className = "group-box";
                groupBox.id = `group-${i}`;

                const label = document.createElement("div");
                label.className = "group-box-label";
                label.textContent = `Group ${i + 1}`;
                groupBox.appendChild(label);

                const itemContainer = document.createElement("div");
                itemContainer.id = `group-items-${i}`;
                groupBox.appendChild(itemContainer);

                groupsDisplay.appendChild(groupBox);
            }
        }

         function updateSpecificGroupBox(groupIndex) {
             const itemContainer = document.getElementById(`group-items-${groupIndex}`);
             if(itemContainer) {
                 const item = document.createElement("div");
                 item.className = "item-block";
                 itemContainer.appendChild(item);
             }
         }

        function setupSimulation() {
            // Get elements again in case they weren't ready before DOM load
             totalInput = totalInput || document.getElementById("dealTotalInput");
             groupsInput = groupsInput || document.getElementById("dealGroupsInput");
             resultValueSpan = resultValueSpan || document.getElementById("resultValue");
             pileCountSpan = pileCountSpan || document.getElementById("pileCount");
             pileDisplay = pileDisplay || document.getElementById("pileDisplay");
             groupsDisplay = groupsDisplay || document.getElementById("groupsDisplay");
             dealBtn = dealBtn || document.getElementById("dealBtn");
             statusMessage = statusMessage || document.getElementById("statusMessage");

             if (!totalInput || !groupsInput || !resultValueSpan || !pileCountSpan || !
                 pileDisplay || !groupsDisplay || !dealBtn || !statusMessage) {
                 console.error("One or more required elements not found during setup!");
                 return;
             }


            initialTotalItems = parseInt(totalInput.value);
            numGroups = parseInt(groupsInput.value);

            if (isNaN(initialTotalItems) || isNaN(numGroups) || numGroups <= 0 ||
                initialTotalItems < 0) {
                statusMessage.textContent = "Please enter valid positive numbers (Groups > 
                    0).";
                dealBtn.disabled = true;
                isDealingComplete = true;
```

```
                 resultValueSpan.textContent = "?";
                 pileCountSpan.textContent = "0";
                 pileDisplay.innerHTML = "";
                 groupsDisplay.innerHTML = "";
                return;
            }

            itemsRemaining = initialTotalItems;
            groupsData = Array(numGroups).fill(0); // Initialize group counts to 0
            nextGroupIndex = 0;
            isDealingComplete = (itemsRemaining === 0); // Complete if starting with 0 items

            statusMessage.textContent = itemsRemaining > 0 ? "Ready␣to␣deal." : "No␣items␣to␣deal.";
            resultValueSpan.textContent = "?";
            updatePileDisplay();
            drawGroupContainers(); // Draw the empty boxes
            dealBtn.disabled = isDealingComplete;
        }

        function dealOneItem() {
            if (!dealBtn || !statusMessage || !resultValueSpan) { // Check elements exist
                 console.error("Button␣or␣status␣element␣not␣found␣during␣deal!");
                 return;
            }

            if (isDealingComplete || itemsRemaining <= 0) {
                statusMessage.textContent = "Dealing␣complete!";
                dealBtn.disabled = true;
                return;
            }

            statusMessage.textContent = ""; // Clear message

            // 1. Decrement remaining items
            itemsRemaining--;

            // 2. Increment count for the target group
            groupsData[nextGroupIndex]++;

            // 3. Visually update pile and target group
            updatePileDisplay();
            updateSpecificGroupBox(nextGroupIndex);

            // 4. Move to next group index (cycle)
            nextGroupIndex = (nextGroupIndex + 1) % numGroups;

            // 5. Check for completion
            if (itemsRemaining === 0) {
                isDealingComplete = true;
                dealBtn.disabled = true;
                statusMessage.textContent = "Dealing␣complete!";
                resultValueSpan.textContent = groupsData[0]; // Show result (items in first group)
            } else {
                statusMessage.textContent = `Dealt 1 item to Group ${nextGroupIndex === 0 ? numGroups : nextGroupIndex}. ${itemsRemaining} left.`;
            }
        }


        // --- Initialize after DOM is loaded ---
        document.addEventListener('DOMContentLoaded', function() {
            // Assign elements to variables now that DOM is ready
            resultValueSpan = document.getElementById("resultValue");
            pileDisplay = document.getElementById("pileDisplay");
            pileCountSpan = document.getElementById("pileCount");
            groupsDisplay = document.getElementById("groupsDisplay");
            dealBtn = document.getElementById("dealBtn");
            statusMessage = document.getElementById("statusMessage");
            totalInput = document.getElementById("dealTotalInput");
            groupsInput = document.getElementById("dealGroupsInput");

            // Now that functions are defined, attach to window if needed by HTML onclick
            // Alternatively, add event listeners here instead of using onclick in HTML
            window.setupSimulation = setupSimulation;
            window.dealOneItem = dealOneItem;


            // Initialize the display on page load
            setupSimulation();

        }); // <<< --- THIS was the likely extra '}' or missing scope boundary ---

    </script>

</div> <!-- End Container -->
</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].

Strategic Multiplicative Reasoning: Division - Inverse of Distributive Reasoning

Compiled by: Theodore M. Savich

March 31, 2025

Transcript

Strategy descriptions and examples adapted from Hackenberg (2025).

- **Teacher:** A man purchases a 56-inch party sub. Each guest at the party receives 8 inches of sub. How many guests can he feed?
- **Student:** I got 7 subs.
- **Teacher:** How did you get 7?
- **Student:** Well I broke 56 inches into 40 inches and 16 inches. I knew that you could make 5 subs with 40 inches, and 2 subs with 16 inches, which would give me a total of 7 subs.

To work on this strategy, it is helpful to list out "easily known multiples" of the known number of items in a group. Then you can use this to build up to the multiple that you don't know.

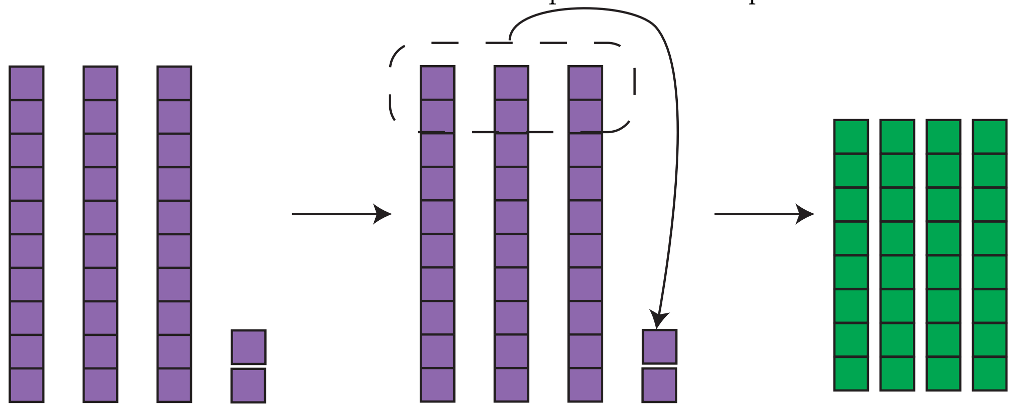
For example, the student likely knew the following:

$$\text{two } 8\text{s} = 16$$
$$\text{five } 8\text{s} = 40$$

He might have also known other 8s, like:

$$\text{three } 8\text{s} = 24$$
$$\text{eight } 8\text{s} = 64$$
$$\text{ten } 8\text{s} = 80$$

But then he used the two 8s and five 8s to help him solve his problem.



$$\begin{aligned}
56 &= ? \times 8\\
56 &= 40 + 16\\
&= \text{five 8s} + \text{two 8s}\\
&= 5 \times 8 + 2 \times 8\\
&= 8(5+2)\\
&= 8 \times 7\\
\text{So, } 56 \div 8 &= 7
\end{aligned}$$

Break the total number of items into multiples that are easier to work with. In other words, view the total as an unknown multiple of a given group size, then express it in terms of familiar or easily calculated multiples. This method essentially involves working backwards, highlighting the fact that division is the inverse of multiplication.

Inverse of the Distributive Property

Strategy Overview

The **Inverse of the Distributive Property** involves reversing the distributive property used in multiplication to aid in solving division problems. This strategy breaks down the total number of items into known multiples, facilitating easier division by calculating the quotient based on these decompositions.

Automaton Design

We design a **Transducing Automaton** (modeled here as a Pushdown Automaton with transduction capabilities) that applies the inverse distributive property by:

- Decomposing the total into known multiples M.
- Calculating the quotient Q by counting the number of times M fits into the total.

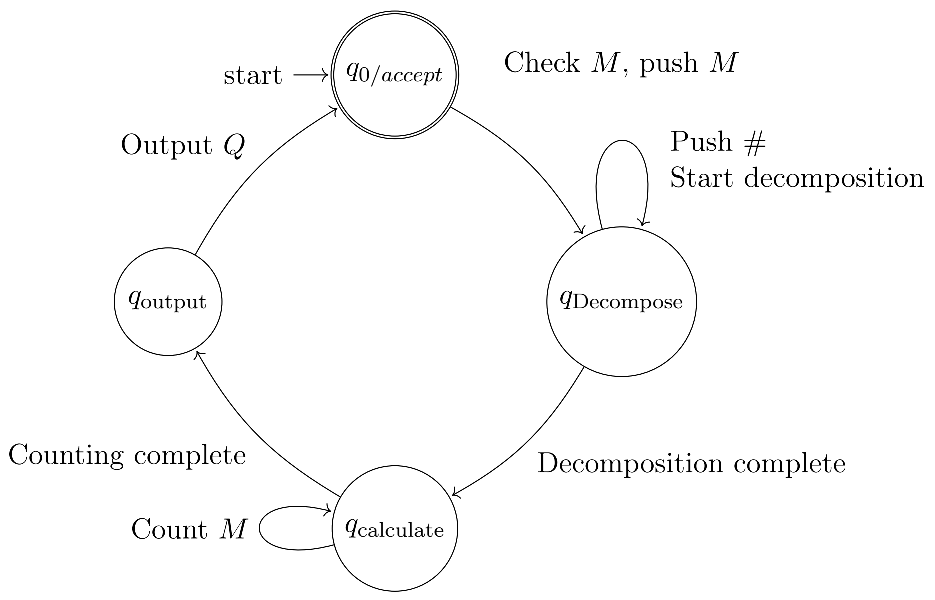
Components of the Automaton

- **States:**
  1. q_{start}: Start state.
  2. $q_{\text{Decompose}}$: Decomposes the total into known multiples.
  3. $q_{\text{calculate}}$: Calculates the quotient by counting multiples.
  4. q_{output}: Outputs the calculated quotient.
- **Input Alphabet:** $\Sigma = \{M\}$, where M represents a known multiple.
- **Stack Alphabet:** $\Gamma = \{\#, Q, M_n\}$:
  - $\#$ is the bottom-of-stack marker.
  - Q represents the quotient.
  - M_n represents an instance of the multiple M decomposed.
- **Initial Stack Symbol:** $\#$

Automaton Behavior

1. **Initialization:**
    - Start in q_{start}; push # onto the stack.
    - Transition to $q_{\text{decompose}}$ to begin decomposition.
2. **Decomposing Total:**
    - In $q_{\text{decompose}}$, for each known multiple M that fits into the remaining total, push M onto the stack.
    - Repeat until the total is fully decomposed.
    - Then transition to $q_{\text{calculate}}$.
3. **Calculating Quotient:**
    - In $q_{\text{calculate}}$, count the number of M symbols on the stack.
    - Push the count as Q onto the stack.
    - Transition to q_{output}.
4. **Outputting the Result:**
    - In q_{output}, read Q from the stack and output it as the quotient.

Circular Automaton Diagram



Example Execution

Problem: Divide 56 items by groups of 8 using the inverse distributive property.

1. **Start:**
    - Stack: #

2. **Decompose:**
    - 56 can be decomposed as 8×7.
    - Push 7 multiples of 8 onto the stack.
3. **Calculate Quotient:**
    - Count the 7 occurrences of M.
    - Push $Q = 7$ onto the stack.
4. **Output:**
    - The automaton outputs 7, meaning 7 groups of 8.

Recursive Handling of Decomposition

The automaton recursively checks for the largest multiple M that fits into the remaining total, ensuring an efficient decomposition and accurate quotient calculation.

HTML Implementation

```
<!DOCTYPE html>
<html>
<head>
    <title>Division: Inverse of Distributive Property</title>
    <style>
        body { font-family: sans-serif; }
        #invDistDiagram { border: 1px solid #d3d3d3; width: 100%; }
        #outputContainer { margin-top: 20px; }
        .diagram-label { font-size: 14px; display: block; margin-bottom: 5px; font-weight:
             bold;}
        .notation-line { margin: 0.2em 0; margin-left: 1em; font-family: monospace;}
        .notation-line.problem { font-weight: bold; margin-left: 0;}
        .notation-step { margin-bottom: 0.5em; }
        /* SVG Styles */
        .total-bar { fill: lightblue; stroke: black; stroke-width: 1; }
        .multiple-segment { stroke: black; stroke-width: 1; }
        .segment-label { font-size: 12px; text-anchor: middle; }
        .factor-label { font-size: 10px; text-anchor: middle; fill: #555; }
        .remainder-segment { fill: lightcoral; stroke: black; stroke-width: 1; }
        .quotient-calc { font-size: 14px; font-weight: bold; }
    </style>
</head>
<body>

<h1>Strategic Multiplicative Reasoning: Division - Inverse of Distributive Property</h1>

<div>
    <label for="invDistTotal">Total (Dividend):</label>
    <input type="number" id="invDistTotal" value="56" min="1"> <!-- Example -->
</div>
<div>
    <label for="invDistGroupSize">Group Size (Divisor):</label>
    <input type="number" id="invDistGroupSize" value="8" min="1"> <!-- Example -->
</div>

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

<div id="outputContainer">
    <h2>Explanation (Notation):</h2>
    <div id="invDistOutput">
        <!-- Text output will be displayed here -->
    </div>
</div>

<h2>Diagram:</h2>
<svg id="invDistDiagram" preserveAspectRatio="xMinYMin␣meet" viewBox="0␣0␣700␣300"></svg>
     <!-- Viewbox for scaling -->


<script>
    // --- Helper SVG Functions ---
    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 drawRect(svg, x, y, width, height, fill, className = '') {
        const rect = document.createElementNS("http://www.w3.org/2000/svg", 'rect');
        rect.setAttribute('x', x); rect.setAttribute('y', y);
        rect.setAttribute('width', Math.max(0, width)); // Ensure width is not negative
        rect.setAttribute('height', height);
        rect.setAttribute('fill', fill);
        rect.setAttribute('class', className);
        svg.appendChild(rect);
    }
    // --- End Helper Functions ---


    // --- Main Inverse Distributive Automaton Function ---
    document.addEventListener('DOMContentLoaded', function() {
        const outputElement = document.getElementById('invDistOutput');
        const totalInput = document.getElementById('invDistTotal');
        const groupSizeInput = document.getElementById('invDistGroupSize');
        const diagramSVG = document.getElementById('invDistDiagram');

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

        window.runInvDistAutomaton = function() {
            try {
                const total = parseInt(totalInput.value);
                const divisor = parseInt(groupSizeInput.value);

                if (isNaN(total) || isNaN(divisor) || total <= 0 || divisor <= 0) {
                    outputElement.textContent = "Please␣enter␣valid␣positive␣numbers";
                    diagramSVG.innerHTML = ’’; return;
                }

                let output = ‘<h2>Inverse of Distributive Property</h2>\n\n‘;
                output += ‘<p class="notation-line␣problem">${total}  ${divisor} = ?</p>\n‘;

                // --- Decomposition Logic ---
                // Define "known" factors (could be dynamic later)
                const knownFactors = [10, 5, 2, 1]; // Prioritize larger factors
                let remainingTotal = total;
                let decomposition = []; // Stores { multiple: M, factor: k }
                let quotientFactors = []; // Stores k values

                output += ‘<p class="notation-line">Decompose ${total} into known multiples of ${divisor}:</p>\n‘;

                while (remainingTotal >= divisor) {
                    let foundMultiple = false;
                    for (const factor of knownFactors) {
                        let multiple = divisor * factor;
                        if (multiple > 0 && multiple <= remainingTotal) {
                            decomposition.push({ multiple: multiple, factor: factor });
                            quotientFactors.push(factor);
                            remainingTotal -= multiple;
                             output += ‘<p class="notation-line␣indent-1">- Found ${multiple} (${factor}  ${divisor}). Remainder: ${remainingTotal}</p>\n‘;
                            foundMultiple = true;
                            break; // Move to next iteration with reduced remainingTotal
                        }
                    }
                     // Safety break if no known multiple fits but remainder >= divisor
                     if (!foundMultiple) {
                         // This might happen if divisor itself is the only option left
                         if (divisor <= remainingTotal) {
                             let factor = 1;
                              let multiple = divisor;
                              decomposition.push({ multiple: multiple, factor: factor });
                              quotientFactors.push(factor);
                              remainingTotal -= multiple;
                              output += ‘<p class="notation-line␣indent-1">- Found ${multiple} (${factor}  ${divisor}). Remainder: ${remainingTotal}</p>\n‘;
                         } else {
                            console.warn("Could␣not␣decompose␣further,␣remainder:", remainingTotal);
                            break; // Exit loop
                         }
                     }
                }

                const quotient = quotientFactors.reduce((sum, factor) => sum + factor, 0);
                const remainder = remainingTotal;

                 output += `<br><p class="notation-line">Sum the factors of the multiples
                     :</p>\n`;
                 output += `<p class="notation-line␣indent-1">${quotientFactors.join('␣+␣')
                     } = ${quotient}</p>\n`;
                 output += `<br><p class="notation-line␣problem">Result: ${quotient}${
                     remainder > 0 ? ` Remainder ${remainder}` : ''}</p>`;


                outputElement.innerHTML = output;
                typesetMath();

                // --- Draw Diagram ---
                drawInverseDistributiveDiagram('invDistDiagram', total, divisor,
                    decomposition, quotient, remainder);

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

        function drawInverseDistributiveDiagram(svgId, total, divisor, decomposition,
            quotient, remainder) {
             const svg = document.getElementById(svgId);
             if (!svg) return;
             svg.innerHTML = '';

             const svgWidth = 700; // Use fixed width from viewBox
             const svgHeight = 300; // Use fixed height from viewBox
             const startX = 30;
             const endX = svgWidth - 30;
             const totalBarY = 50;
             const totalBarHeight = 30;
             const decompBarY = totalBarY + totalBarHeight + 40;
             const decompBarHeight = 30;
             const labelOffsetY = -10; // Above bars
             const factorLabelOffsetY = 15; // Below decomp bars

             // --- Scaling ---
             const availableWidth = endX - startX;
             const scale = availableWidth / total; // Scale based on total value

             // --- Draw Total Bar ---
             createText(svg, startX, totalBarY + labelOffsetY, `Total: ${total}`, 'diagram
                 -label');
             drawRect(svg, startX, totalBarY, total * scale, totalBarHeight, 'lightblue',
                 'total-bar');

             // --- Draw Decomposition Segments ---
             createText(svg, startX, decompBarY + labelOffsetY, `Decomposition into Multiples of ${divisor}`);
             let currentX = startX;
             decomposition.forEach(part => {
                 const segmentWidth = part.multiple * scale;
                 drawRect(svg, currentX, decompBarY, segmentWidth, decompBarHeight, `hsl(${part.factor * 25}, 70%, 70%)`, 'multiple-segment'); // Vary color by factor
                 // Label with the multiple value
                 createText(svg, currentX + segmentWidth / 2, decompBarY + decompBarHeight / 2 + 5, `${part.multiple}`, 'segment-label', 'middle');
                  // Label with the multiplication fact
                  createText(svg, currentX + segmentWidth / 2, decompBarY + decompBarHeight + factorLabelOffsetY, `(${part.factor}  ${divisor})`, 'factor-label', 'middle');
                 currentX += segmentWidth;
             });

             // --- Draw Remainder Segment ---
             if (remainder > 0) {
                 const segmentWidth = remainder * scale;
                  drawRect(svg, currentX, decompBarY, segmentWidth, decompBarHeight, 'lightcoral', 'remainder-segment');
                  createText(svg, currentX + segmentWidth / 2, decompBarY + decompBarHeight / 2 + 5, `${remainder}`, 'segment-label', 'middle');
                  createText(svg, currentX + segmentWidth / 2, decompBarY + decompBarHeight + factorLabelOffsetY, `(Rem)`, 'factor-label', 'middle');
                  currentX += segmentWidth;
             }

             // --- Display Quotient Calculation ---
              let quotientY = decompBarY + decompBarHeight + factorLabelOffsetY + 40;
              createText(svg, startX, quotientY, `Quotient = ${decomposition.map(p => p.factor).join(' + ')} = ${quotient}`, 'quotient-calc');


             // --- Adjust ViewBox ---
              // No need to adjust height dynamically for this layout if 300 is enough
              // svg.setAttribute('viewBox', `0 0 ${svgWidth} ${svgHeight}`);
         }

         function typesetMath() { /* Placeholder */ }

         // Initialize on page load
         runInvDistAutomaton();

     }); // End DOMContentLoaded
 </script>

 </body>
 </html>
```

References

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

Division Strategies - Strategic Trials

Compiled by: Theodore M. Savich

March 31, 2025

This is a sharing division strategy. With sharing division problems, the number of items in each group is unknown, while the number of groups and the total number of items are both known.

$\boxed{\text{Number of groups}} \times \boxed{\text{Unknown Number of items in each group}} = \boxed{\text{Total number of items}}$

Transcript

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

- **Teacher:** Mrs. Carpenter made 56 cupcakes for a birthday party. She has eight boxes to carry the cupcakes to his party. How many cupcakes should she put in each box if she wants to put the same number of cupcakes in each box?
- **Student:** [inaudible] Put seven in. Seven.
- **Teacher:** I can tell just tell you did that. Thank you very much, Victoria.

This strategy is more sophisticated than Dealing by Ones because it involves selecting an initial, reasonable group size, testing it, and then logically refining that choice as needed.
Description of Strategic Trials:
Begin with an initial trial number for the items per group. **Utilize a multiplication strategy** to calculate the total number of items and verify it against the given total. Adjust your trial number upward or downward as necessary, and recalculate until you arrive at the correct result.
Notation and Visual Representations for Strategic Trials: Use clear notation and diagrams to illustrate the equal groups multiplication strategy you have chosen.

For example, second-grade student Victoria was tasked with determining how many cupcakes should be placed in each of 8 boxes, given a total of 56 cupcakes. She initially assumed 8 cupcakes per box and employed a doubling method to compute the total:

$$8 + 8 = 16$$

$$16 + 16 = 32$$

$$32 + 32 = 64$$

Seeing that 64 exceeded the given total, she then tried 6 cupcakes per box:

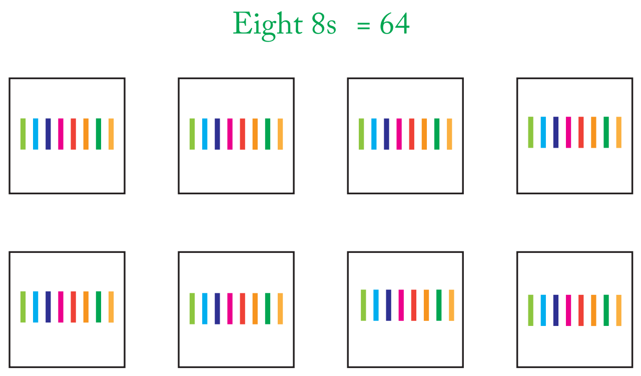
$$6 + 6 = 12$$

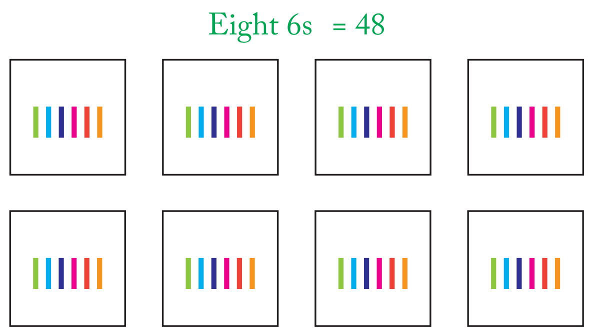
$$12 + 12 = 24$$

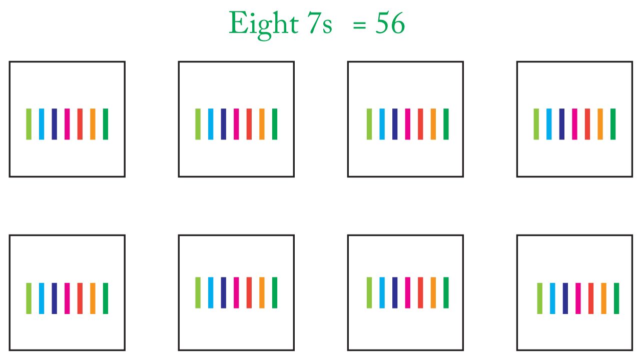
$$24 + 24 = 48$$

Realizing 48 was too low, Victoria understood she was estimating the number of cupcakes per box. After trying 8 (which was too high) and 6 (which was too low), she decided to test 7 cupcakes per box:

$$\begin{aligned} 7 + 7 &= 14 \\ 14 + 14 &= 28 \\ 28 + 28 &= 56 \quad \text{(using her addition strategy)} \end{aligned}$$







She concluded that each box should contain 7 cupcakes. In class, we highlighted that her method was not merely "trial and error," but a thoughtful process of strategic adjustment. When the initial guess was too high, she adjusted downward, and when it was too low, she adjusted upward. This iterative process is a hallmark of strategic trials.

Strategic Trials

Strategy Overview

Strategic Trials involves testing different grouping configurations to find the correct division outcome. This strategy is iterative and relies on trial-and-error to determine the appropriate number of groups or the group size required for division.

Automaton Design

We design a **Pushdown Automaton (PDA)** that systematically:

1. Attempts a trial grouping by pushing a trial marker T and assigning a set of elements.
2. Checks whether the trial group meets the required size.
3. Adjusts the trial group if the size is incorrect.
4. Upon a correct trial, confirms the group by pushing a group identifier G and then outputs the final grouping.

Automaton Tuple

The PDA is defined as the 7-tuple

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

where:

- $Q = \{q_{0/accept}, q_{\text{trial}}, q_{\text{check}}, q_{\text{adjust}}, q_{\text{output}}\}$ is the set of states. (Here, $q_{0/accept}$ serves as both the start and the accepting state.)
- $\Sigma = \{E\}$ is the input alphabet (with E representing an element).
- $\Gamma = \{\#, T, G\}$ is the stack alphabet:
  - $\#$ is the bottom-of-stack marker.
  - T represents a trial grouping.
  - G represents a confirmed group.
- $q_{0/accept}$ is the start (and accept) state.
- $\#$ is the initial stack symbol.
- $F = \{q_{0/accept}\}$ is the set of accepting states.

State Transition Table

Current State	Input Symbol	Stack Top	Next State	Stack Operation	Description
$q_{0/accept}$	ε	—	q_{trial}	Push #	Initialize
q_{trial}	ε	any	q_{check}	Push T; assign a trial group	Attempt trial
q_{check}	ε	any	q_{output}	(If trial correct: push G)	Trial correct
q_{check}	ε	any	q_{adjust}	—	Trial incorrect
q_{adjust}	ε	any	q_{trial}	Adjust trial	Modify trial group
q_{output}	ε	any	$q_{0/accept}$	Count G's	Output final grouping

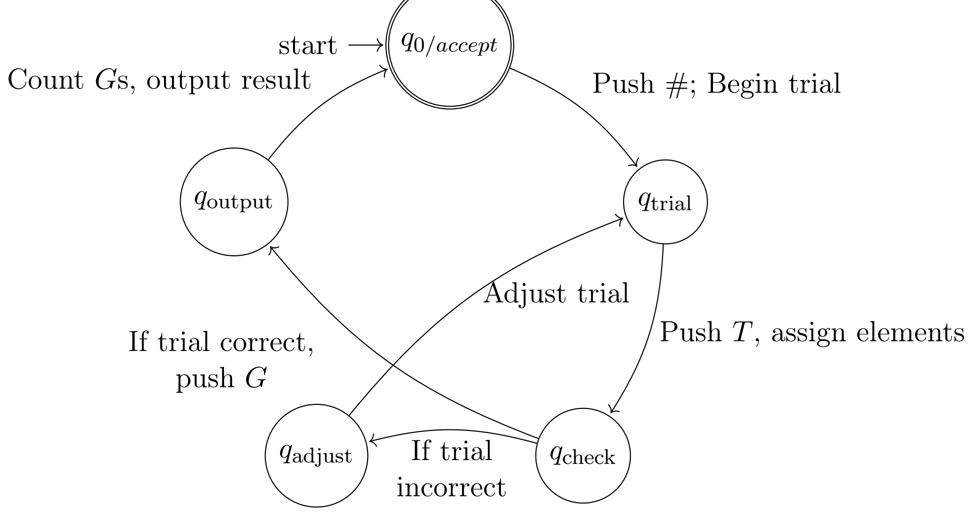
Automaton Behavior

1. **Initialization:**
   - Start in $q_{0/accept}$, push # onto the stack.
   - Transition to q_{trial} to begin the trial process.
2. **Attempting a Trial:**

- In q_{trial}, push T to represent a trial group and assign a set of elements to it.
- Transition to q_{check}.

3. **Checking the Trial:**
    - In q_{check}, evaluate if the trial group meets the required size.
    - If the trial is correct, push a confirmed group G and transition to q_{output}.
    - If the trial is incorrect, transition to q_{adjust}.
4. **Adjusting the Trial:**
    - In q_{adjust}, modify the trial group size (by adding or removing elements).
    - Return to q_{trial} to try again.
5. **Outputting the Result:**
    - In q_{output}, count the number of confirmed groups (G symbols) on the stack.
    - Output the final grouping and transition back to $q_{0/accept}$ (the merged start/accept state).

Circular PDA Diagram



Example Execution

Problem: Divide 24 items into groups of 8 using strategic trials.

1. **Start:**
    - The initial stack contains: # followed by 24 E symbols.
2. **Trial 1:**
    - In q_{trial}, a trial group of 7 elements is attempted (push T, assign 7 E symbols).
    - In q_{check}, the trial is evaluated: $7 \neq 8$, so transition to q_{adjust}.
3. **Adjust Trial:**

- In q_{adjust}, the trial is modified (e.g., increase group size to 8).
- Return to q_{trial} for a new attempt.

4. **Trial 2:**
   - In q_{trial}, attempt a trial group of 8 elements.
   - In q_{check}, the trial is correct ($8 = 8$); a confirmed group G is pushed.

5. **Repeat:**
   - Continue trials until all 24 items are grouped.
   - Final output: 3 groups of 8.

Iterative Handling of Trials

The PDA iteratively attempts different group sizes, adjusting the trial configuration as needed based on feedback from the check phase. This iterative process continues until the correct grouping is achieved, ensuring an accurate division.

HTML Implementation

```html
<!DOCTYPE html>
<html>
<head>
    <title>Division: Strategic Trials</title>
    <style>
        body { font-family: sans-serif; }
        .container { max-width: 800px; margin: 10px auto; padding: 10px;}
        .control-section, .trials-section, .result-section {
             margin-bottom: 20px; padding: 10px; border: 1px solid #eee;
             background-color: #f9f9f9; border-radius: 5px;
        }
        label { margin-right: 5px;}
        input[type=number] { width: 60px; margin-right: 15px;}
        button { padding: 5px 10px; font-size: 1em; margin-right: 5px; }
        #statusMessage { color: #e65c00; font-weight: bold; margin-left: 15px;}

        .trial-visualization {
            margin-top: 15px;
            padding-top: 10px;
            border-top: 1px dashed #ccc;
        }
         .group-container { /* Container for all groups in a trial */
             display: flex;
             flex-wrap: wrap; /* Allow groups to wrap */
             gap: 10px; /* Space between groups */
             margin-bottom: 5px;
         }
        .group-box {
             display: inline-block; /* Display groups inline */
             border: 1px solid #999;
             padding: 4px;
             background-color: #e8f4ff;
             min-width: 40px; /* Minimum width */
             text-align: center; /* Center items */
         }
         .group-box-label { font-size: 0.8em; color: #555; margin-bottom: 3px; display:
             block;}
        .item-block {
            display: inline-block; /* Items side-by-side */
            width: 8px; height: 8px; margin: 1px; /* Smaller items */
            background-color: #6495ED; /* Cornflower blue */
            border: 1px solid #444;
        }
        .trial-summary { font-weight: bold; margin-top: 5px; }
        .trial-correct { color: darkgreen; }
        .trial-incorrect { color: darkred; }
        #finalResultValue { font-size: 1.5em; font-weight: bold; color: darkgreen; }
    </style>
</head>
<body>
<div class="container">

```

```
    <h1>Division Strategies - Strategic Trials</h1>

    <div class="control-section">
        <label for="stratTotalInput">Total Items:</label>
        <input type="number" id="stratTotalInput" value="56" min="1"> <!-- Example -->
        <label for="stratGroupsInput">Number of Groups:</label>
        <input type="number" id="stratGroupsInput" value="8" min="1"> <!-- Example -->
        <button onclick="setupTrialSimulation()">Set Up / Reset</button>
        <button onclick="performNextTrial()" id="trialBtn" disabled>Perform Next Trial</
            button>
         <span id="statusMessage"></span>
    </div>

    <div class="trials-section">
         <strong>Trials:</strong>
         <div id="trialsDisplay">
             <!-- Trial visualizations will be added here -->
         </div>
    </div>

     <div class="result-section">
        <strong>Result (Items per group):</strong> <span id="finalResultValue">?</span>
    </div>


    <script>
        // --- Simulation State Variables ---
        let totalItems = 0;
        let numGroups = 0;
        let currentTrialSize = -1; // -1 indicates simulation not started or needs initial
             guess
        let attempts = []; // Stores history: { trialSize: number, trialResult: number,
            outcome: string }
        let finalGroupSize = null; // The correct answer when found
        let isTrialComplete = true;

        // --- DOM Element References ---
        const totalInput = document.getElementById("stratTotalInput");
        const groupsInput = document.getElementById("stratGroupsInput");
        const finalResultValueSpan = document.getElementById("finalResultValue");
        const trialsDisplay = document.getElementById("trialsDisplay");
        const trialBtn = document.getElementById("trialBtn");
        const statusMessage = document.getElementById("statusMessage");

        // --- Simulation Functions ---
        function setupTrialSimulation() {
            totalItems = parseInt(totalInput.value);
            numGroups = parseInt(groupsInput.value);

            if (isNaN(totalItems) || isNaN(numGroups) || numGroups <= 0 || totalItems < 0)
                 {
                statusMessage.textContent = "Please␣enter␣valid␣positive␣numbers␣(Groups␣>␣
                    0).";
                trialBtn.disabled = true;
                isTrialComplete = true;
                finalResultValueSpan.textContent = "?";
                trialsDisplay.innerHTML = ""; // Clear previous trials
                return;
            }

            // Make the first guess intentionally off (e.g., +/- 1 or 2 from rough
                estimate)
            let roughEstimate = Math.max(1, Math.round(totalItems / numGroups)); // Ensure
                 guess is at least 1
            let randomOffset = Math.random() < 0.5 ? (roughEstimate > 1 ? -1 : 1) : 1; //
                Offset by +/- 1
            currentTrialSize = roughEstimate + randomOffset;
            // Ensure guess isn't accidentally correct if estimate was close
            if (currentTrialSize * numGroups === totalItems && currentTrialSize > 1) {
                currentTrialSize--; // Adjust if first guess happens to be right
            }
             if (currentTrialSize <= 0) currentTrialSize = 1; // Ensure guess is at least
                 1


            attempts = []; // Clear history
            finalGroupSize = null;
            isTrialComplete = false;

            statusMessage.textContent = `Ready. Initial trial guess: ${currentTrialSize}
                items per group.`;
            finalResultValueSpan.textContent = "?";
            trialsDisplay.innerHTML = ""; // Clear previous trials visually
            trialBtn.disabled = false;
        }

        function performNextTrial() {
            if (isTrialComplete) {
                statusMessage.textContent = "Found correct group size! Press Reset to start
                     again.";
                trialBtn.disabled = true;
                return;
            }

            statusMessage.textContent = `Trying ${currentTrialSize} items per group...`;

            // 1. Multiply to get trial total
            const trialResult = currentTrialSize * numGroups;

            // 2. Check against actual total
            let outcome = "";
            let outcomeClass = "";
            if (trialResult === totalItems) {
                outcome = "Correct!";
                outcomeClass = "trial-correct";
                finalGroupSize = currentTrialSize;
                isTrialComplete = true;
                trialBtn.disabled = true; // Disable button once correct
                statusMessage.textContent = `Found correct group size: ${finalGroupSize
                    }!`;
                finalResultValueSpan.textContent = finalGroupSize;
            } else if (trialResult < totalItems) {
                outcome = `Too Low (${trialResult} < ${totalItems})`;
                outcomeClass = "trial-incorrect";
            } else { // trialResult > totalItems
                outcome = `Too High (${trialResult} > ${totalItems})`;
                 outcomeClass = "trial-incorrect";
            }

            // 3. Store attempt
            attempts.push({
                trialSize: currentTrialSize,
                trialResult: trialResult,
                outcome: outcome,
                outcomeClass: outcomeClass
            });

             // 4. Draw this attempt
             drawTrialVisualization(currentTrialSize, numGroups, trialResult, outcome,
                 outcomeClass);


            // 5. Adjust for next trial (if not correct)
            if (!isTrialComplete) {
                if (trialResult < totalItems) {
                     // Increase guess (could be smarter, e.g., based on how far off)
                     currentTrialSize++;
                } else {
                     // Decrease guess
                     currentTrialSize--;
                     if (currentTrialSize <= 0) currentTrialSize = 1; // Don't guess 0 or
                         negative
                }
                 statusMessage.textContent += ` Adjusting guess to ${currentTrialSize}.`;
            }
        }

        function drawTrialVisualization(trialSize, groups, result, outcome, outcomeClass)
            {
            const trialDiv = document.createElement('div');
            trialDiv.className = 'trial-visualization';

            const groupContainer = document.createElement('div');
            groupContainer.className = 'group-container';

            for (let g = 0; g < groups; g++) {
                const groupBox = document.createElement("div");
                groupBox.className = "group-box";
                // groupBox.innerHTML = `<span class="group-box-label">Group ${g + 1}</span
                    >`; // Optional label

                 // Arrange items within the box (e.g., simple horizontal flow)
                 let itemsHtml = '';
                 let itemsPerRow = Math.max(5, Math.ceil(Math.sqrt(trialSize))); // Simple layout heuristic
                 for(let i = 0; i < trialSize; i++) {
                     itemsHtml += `<span class="item-block"></span>`;
                     if ((i + 1) % itemsPerRow === 0) itemsHtml += '<br>'; // Add line break
                 }
                 groupBox.innerHTML += itemsHtml;
                 groupContainer.appendChild(groupBox);
            }
            trialDiv.appendChild(groupContainer);

             const summary = document.createElement('div');
             summary.className = 'trial-summary';
             summary.innerHTML = `Trial: ${groups} groups  ${trialSize} items/group = ${result}. <span class="${outcomeClass}">${outcome}</span>`;
             trialDiv.appendChild(summary);


            trialsDisplay.appendChild(trialDiv);
            trialsDisplay.scrollTop = trialsDisplay.scrollHeight; // Scroll to bottom
        }


        // --- Helper SVG/Typeset Functions (Not needed for this block viz) ---
        function typesetMath() { /* Placeholder */ }

        // --- Initialize ---
        setupTrialSimulation(); // Initialize state on load


</script>

</div> <!-- End Container -->
</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].

Division Strategies - Using Commutative Reasoning

Compiled by: Theodore M. Savich

March 31, 2025

Strategy descriptions and examples adapted from Hackenberg (2025).

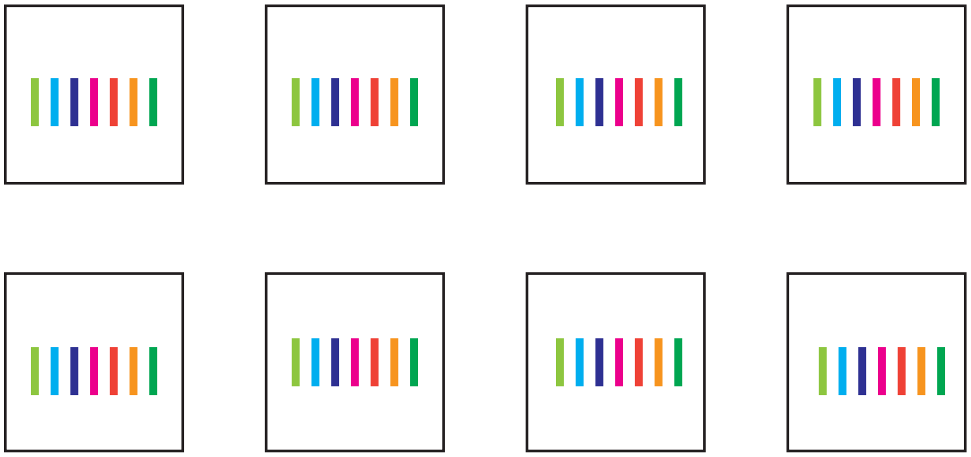
This is a strategy for transforming the context of a sharing division problem (where the number of items in each group is unknown) into one where measurement division strategies can be used. Measurement division strategies are generally easier to use because students can count something.

$$\boxed{\text{Number of groups}} \times \boxed{\text{Unknown Number of items in each group}} = \boxed{\text{Total number of items}}$$

The idea of **Using Commutative Reasoning** is to reframe N —- the number of groups -— as the act of placing one item into each group simultaneously. So, when you count one N, you're putting one item into every group; counting three Ns means each group receives three items. This new interpretation of N enables us to apply measurement division strategies, since our goal becomes finding how many times N fits into the total number of items. This method is incredibly useful—but first, we need to clarify this shift in how we view N! You might create a chart that illustrates distributing items one round at a time across the groups as you count by N. When you're learning this strategy, using such visual representations can be very beneficial. The problem remains a sharing division problem, but we can now effectively apply measurement division strategies to solve it.

Example: There are 56 cupcakes and 8 boxes. If we are going to put an equal number of cupcakes in each box, how many will go in each box?

The original meaning of 8 in the problem is # of boxes, or # of groups. The meaning Victoria gave to 8 when she wrote down eight 8s (see above) was that 8 meant the # of items in a group. Neither of these meanings for 8 would allow her to count by repeatedly by 8 until she reaches 56, and then to know she has solved the problem. In other words, neither of these meanings for 8 will allow her to count seven 8s as a meaningful solution to the problem. WHY?



Number of cupcakes given out	Number of cupcakes in each box
One 8 = 8	1
Two 8s = 16	2
Three 8s = 24	3
Four 8s = 32	4
Five 8s = 40	5
Six 8s = 48	6
Seven 8s = 56	7

Using Commutative Reasoning

Strategy Overview

Using Commutative Reasoning leverages the commutative property of multiplication to facilitate division. By repackaging the number of groups and the number of items in each group, this strategy simplifies the division process and aligns it with multiplication reasoning.

Automaton as a 7-Tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \#, F),$$

where:

- $Q = \{\, q_0,\ q_{\text{read}},\ q_{\text{calculate}},\ q_{\text{output}},\ q_{\text{accept}} \}$.
- $\Sigma = \{G, E\}$ is the input alphabet (G = group information, E = total items).
- $\Gamma = \{\#, G, E, Q\}$ is the stack alphabet, with $\#$ the bottom marker.
- q_0 is the start state;
- $\#$ is the initial stack symbol.

- $F = \{q_{\text{accept}}\}$ is the set of accepting states.

State Transition Table (Corrected)

Current State	**Input Symbol**	**Stack Top**	**Next State**	**Stack Operation**	**Action / Interpretation**
q_0	ε	(empty)	q_{read}	Push #	Initialize stack with #.
q_{read}	G	#	q_{read}	Push G	Read group info.
q_{read}	E	G	$q_{\text{calculate}}$	Push E	Read total elements.
$q_{\text{calculate}}$	ε	E	q_{output}	Pop E, Pop G, Push $Q = E/G$	Perform division $E \div G$.
q_{output}	ε	Q	q_{accept}	Output Q	Show result (quotient).
q_{accept}	ε	#	q_{accept}	No change	Accept.

Automaton Behavior (Step-by-Step)

1. **Initialization:** In state q_0, push the bottom-of-stack marker #, then move to q_{read}.
2. **Reading the Inputs:**
   - Reading G (e.g. 8): push G onto the stack.
   - Reading E (e.g. 56): push E onto the stack, then move to $q_{\text{calculate}}$.
3. **Calculation:** In $q_{\text{calculate}}$, pop both E and G, compute the quotient $Q = \frac{E}{G}$, and push Q.
4. **Output:** Transition to q_{output}, output Q, then move to q_{accept} to finish.

Corrected Example Execution

Problem: Divide 56 items into 8 groups.

1. **Inputs Read:**
$$G = 8, \quad E = 56.$$
2. **Stored on Stack:** # at the bottom, then G, then E.
3. **Calculation Step:**
$$Q = \frac{E}{G} = \frac{56}{8} = 7.$$
4. **Output:** The automaton pushes $Q = 7$ and transitions to q_{output}.

No contradictory "$\frac{8}{56}$" arises here, because we never literally swap the roles of G and E. Instead, the "commutative" viewpoint is *conceptual*: we regard "8 groups" as "counting by eights" out of 56, which is the usual measurement-division approach.

HTML Implementation

References

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