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

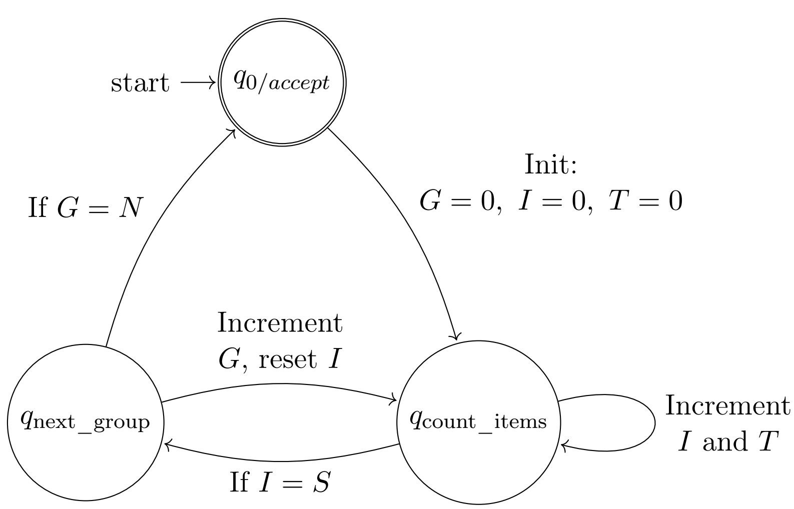

Figure 1: FSA with counters to coordinate item and group counting by ones.

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].