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:

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.

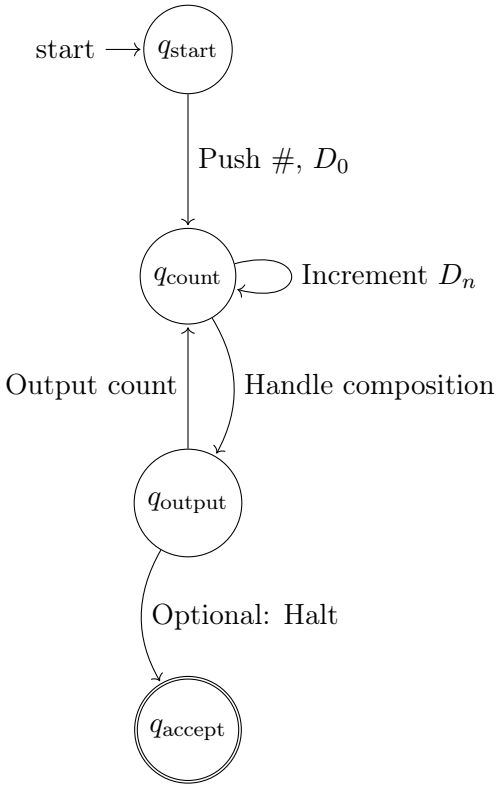

Figure 1: Conceptual State Diagram for Recursive Counting with a PDA

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., $999\ldots9$), new stack symbols are pushed to extend the number (e.g., $1000\ldots0$).

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;
```

```
198
199             // Draw tens groups using SVG
200             for (let t = 0; t < tallyTens; t++) {
201                 const isToggleable = (count === 10 && t === 0); // Clickable only at count
                        10
202                 const useSlashed = isToggleable ? tenAsSlashTally : tenAsSlashTally; //
                        Draw based on toggle state
203                 drawTallyGroupSVG(tallyContainer, useSlashed, isToggleable);
204
205                  // No extra spacer needed, margin on SVG handles it
206             }
207
208             // Draw remainder (ones) tallies as simple text |
209             if (tallyOnes > 0) {
210                 const onesSpan = document.createElement("span");
211                 onesSpan.className = "tally-mark";
212                 onesSpan.textContent = "|".repeat(tallyOnes);
213                 tallyContainer.appendChild(onesSpan);
214             }
215
216         } // End of updateDisplay
217
218         // Initialize the display on page load
219         updateDisplay();
220
221     </script>
222
223 </body>
224 </html>
```