Counting in Base 10

Theodore M. Savich

April 4, 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 a PDA for Two-Digit Base-10 Counting (0-99)

While the concept of recursive counting is powerful, implementing it directly with standard Pushdown Automaton (PDA) transitions faces challenges, particularly in managing the carry operation across an arbitrary number of digits simultaneously. Standard PDAs can only inspect the top symbol of the stack, making it difficult to modify a lower digit based on a carry signal from the top digit while preserving the stack structure easily.

Therefore, we will adapt the approach to model a more constrained, yet practical, automaton: a **two-digit base-10 counter** capable of representing numbers from 0 to 99. This model demonstrates how place value and carries can be managed using distinct stack symbols and specific state transitions within the PDA framework.

4.1 Simulating Two Digits on One Stack

We simulate the two place values (Tens and Units) using distinct symbols on the PDA's single stack:

- **Units Digit Symbols:** $U_0, U_1, \ldots, U_9$
- **Tens Digit Symbols:** $T_0, T_1, \ldots, T_9$

A number, represented conventionally as XY (where X is the tens digit and Y is the units digit), will be stored on the stack with the units digit on top. The stack configuration for the number XY will be $(\#, T_X, U_Y)$, where $\#$ is the bottom marker. For example, the number 23 would be represented by the stack $(\#, T_2, U_3)$.

4.2 Components of the Two-Digit PDA

The PDA is defined by the following components ($M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, Z_0, F)$):

- **States (Q):** A finite set of states is required to manage the counting and carry logic:
  1. q_{start}: The initial state.
  2. q_{idle}: The main state where the PDA resides when holding a valid count (0-99). This is the primary accepting state.
  3. $q_{\text{inc\_tens}}$: An intermediate state entered when the units digit rolls over from 9 to 0, signaling that the tens digit needs processing via an epsilon transition.
  4. q_{halt}: A non-accepting state entered when an attempt is made to increment the count beyond 99 (overflow).
- **Input Alphabet (Σ):** Contains a symbol representing one unit to be counted, plus the empty string ϵ for internal transitions. Let's use 'tick' for the input unit.
$$\Sigma = \{\text{tick}, \epsilon\}$$
- **Stack Alphabet (Γ):** Includes the bottom marker and symbols for each digit in each place value.
$$\Gamma = \{\#, T_0, \ldots, T_9, U_0, \ldots, U_9\}$$
- **Transition Function (δ):** Defined in Section 8.
- **Initial State (q_0):** q_{start}.
- **Initial Stack Symbol (Z_0):** $\#$.
- **Final States (F):** Only the state representing a valid count is accepting.
$$F = \{q_{\text{idle}}\}$$

4.3 Automaton Behavior

The two-digit counter operates as follows:

1. **Initialization:** Start in q_{start}. On an epsilon transition, push T_0 then U_0 onto the stack (representing 0) and transition to q_{idle}. Stack: $(\#, T_0, U_0)$.
2. **Counting (Units Increment):** In state q_{idle}, read a 'tick' input.
   - If the top symbol is U_n where $n < 9$, pop U_n, push U_{n+1}, and remain in q_{idle}.
   - If the top symbol is U_9, pop U_9 (do not push U_0 yet). Transition to $q_{\text{inc\_tens}}$ to handle the carry. The T_X symbol is now exposed on top of the stack.
3. **Carry Handling (Tens Increment):** In state $q_{\text{inc\_tens}}$, perform an epsilon transition based on the exposed tens digit T_m:
   - If the top symbol is T_m where $m < 9$, pop T_m. Push T_{m+1}, then push U_0. Transition back to q_{idle}. The carry is complete. Stack: $(\#, T_{m+1}, U_0)$.
   - If the top symbol is T_9 (representing an attempt to increment 99), pop T_9. Push T_0, then push U_0. Transition to the non-accepting state q_{halt}. Stack: $(\#, T_0, U_0)$.
4. **Halt State:** Once in q_{halt}, no further transitions are defined. The machine halts and rejects any further input, indicating overflow.

5 State Diagram for Two-Digit Counter

The diagram below illustrates the states and transitions for the two-digit (0-99) counter PDA. Note that multi-symbol stack operations are abbreviated (e.g., $T_m \to U_0, T_{m+1}$ means pop T_m, push T_{m+1}, push U_0).

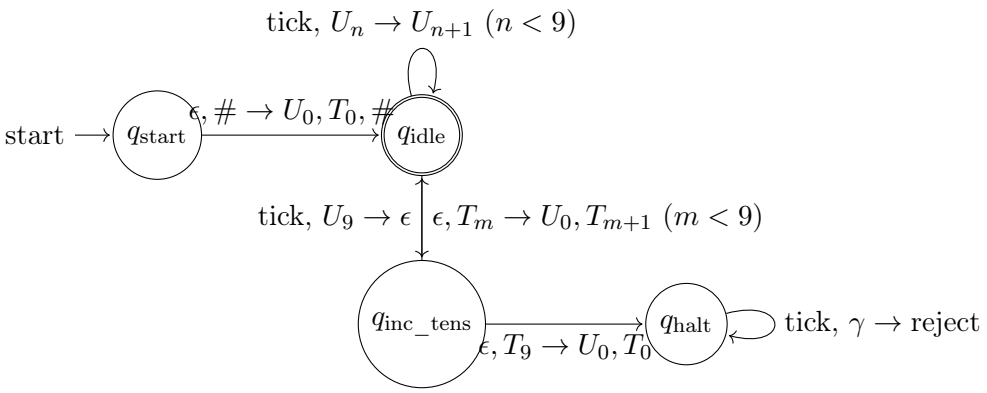

Figure 1: State Diagram for the Two-Digit (0-99) Counter PDA

6 Detailed Example Execution: Counting from 8 to 11

This section demonstrates how the two-digit PDA counts from 8 to 11.

1. **Start at 8:** Assume the PDA has processed 8 'tick' inputs and is in state q_{idle} with stack $(\#, T_0, U_8)$.
2. **Input 9 ('tick'):**
   - In state q_{idle}, reads 'tick'. Top symbol is U_8.
   - Transition: $\delta(q_{\text{idle}}, \text{tick}, U_8) = (q_{\text{idle}}, (U_9))$.
   - Pops U_8, pushes U_9. Remains in q_{idle}.
   - Stack: $(\#, T_0, U_9)$ (represents 9).
3. **Input 10 ('tick'):**
   - In state q_{idle}, reads 'tick'. Top symbol is U_9.
   - Transition: $\delta(q_{\text{idle}}, \text{tick}, U_9) = (q_{\text{inc\_tens}}, ())$.
   - Pops U_9, pushes nothing. Enters state $q_{\text{inc\_tens}}$.
   - Stack: $(\#, T_0)$.
   - Now, an epsilon transition occurs from $q_{\text{inc\_tens}}$. Top symbol is T_0.
   - Transition: $\delta(q_{\text{inc\_tens}}, \epsilon, T_0) = (q_{\text{idle}}, (U_0, T_1))$.
   - Pops T_0. Pushes T_1, then pushes U_0. Enters state q_{idle}.
   - Stack: $(\#, T_1, U_0)$ (represents 10).
4. **Input 11 ('tick'):**
   - In state q_{idle}, reads 'tick'. Top symbol is U_0.

- Transition: $\delta(q_{\text{idle}}, \text{tick}, U_0) = (q_{\text{idle}}, (U_1))$.
- Pops U_0, pushes U_1. Remains in q_{idle}.
- Stack: $(\#, T_1, U_1)$ (represents 11).

7 Handling the Carry from Units to Tens

The crucial step in base-10 counting is handling the carry operation. In this two-digit PDA, the carry from the units place (when incrementing 9, 19, 29, etc.) to the tens place is managed by a combination of state change and stack manipulation:

1. **Triggering the Carry:** When the units digit U_9 is incremented in state q_{idle}, it is popped from the stack, but the new units digit U_0 is *not* immediately pushed. Instead, the PDA transitions to the dedicated carry state $q_{\text{inc\_tens}}$.
2. **Exposing the Tens Digit:** Popping U_9 leaves the tens digit symbol (T_m) exposed as the top stack symbol.
3. **Processing the Tens Digit:** State $q_{\text{inc\_tens}}$ performs an immediate epsilon transition based on the exposed T_m:
   - If $m < 9$, T_m is popped. The incremented tens digit T_{m+1} is pushed first, followed by the new units digit U_0. The PDA returns to q_{idle}, having completed the carry (e.g., state 19 becomes 20).
   - If $m = 9$, T_9 is popped. T_0 is pushed, followed by U_0. The PDA transitions to q_{halt}, indicating an overflow beyond 99.

This mechanism correctly updates the tens place value using an intermediate state and ensures the stack correctly represents the new two-digit number after the carry.

8 Formal Transition Function (δ) for Two-Digit Counter

The transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$ maps the current state, input symbol (or epsilon), and top stack symbol to a new state and a sequence of symbols to push onto the stack (replacing the popped top symbol). Note: (S_1, S_2) means push S_2 then push S_1.

- **Initialization:**
$$\delta(q_{\text{start}}, \epsilon, \#) = (q_{\text{idle}}, (U_0, T_0, \#))$$
- **Idle State (Incrementing Units):** For $n \in \{0, 1, \ldots, 8\}$:
$$\delta(q_{\text{idle}}, \text{tick}, U_n) = (q_{\text{idle}}, (U_{n+1}))$$
$$\delta(q_{\text{idle}}, \text{tick}, U_9) = (q_{\text{inc\_tens}}, ())$$
- **Carry State (Incrementing Tens via Epsilon):** For $m \in \{0, 1, \ldots, 8\}$:
$$\delta(q_{\text{inc\_tens}}, \epsilon, T_m) = (q_{\text{idle}}, (U_0, T_{m+1}))$$
$$\delta(q_{\text{inc\_tens}}, \epsilon, T_9) = (q_{\text{halt}}, (U_0, T_0))$$
- **Halt State:** No transitions defined out of q_{halt}.
$$\delta(q_{\text{halt}}, \cdot, \cdot) = \emptyset$$

9 Example: Counting from 98 to Overflow

This example shows how the PDA handles the count reaching and exceeding 99.

1. **Start at 98:** Assume the PDA is in state q_{idle} with stack $(\#, T_9, U_8)$.
2. **Input 99 ('tick'):**
   - In q_{idle}, reads 'tick'. Top is U_8.
   - Pops U_8, pushes U_9. Stays in q_{idle}.
   - Stack: $(\#, T_9, U_9)$ (represents 99).
3. **Input 100 ('tick'):**
   - In q_{idle}, reads 'tick'. Top is U_9.
   - Pops U_9, pushes nothing. Enters state $q_{\text{inc\_tens}}$.
   - Stack: $(\#, T_9)$.
   - Epsilon transition from $q_{\text{inc\_tens}}$. Top is T_9.
   - Pops T_9. Pushes T_0, then pushes U_0. Enters state q_{halt}.
   - Stack: $(\#, T_0, U_0)$.
4. **State q_{halt}:** The PDA is now in the non-accepting state q_{halt}. It has successfully represented the rollover conceptually (stack shows 00), but the state indicates the limit was exceeded. Any further 'tick' inputs will cause the machine to remain stuck in q_{halt} as no transitions are defined, leading to rejection.

10 Practical Considerations and Limitations

This two-digit counter PDA demonstrates a successful application of the model for a fixed range.

- **Fixed Range:** The primary limitation is the fixed range (0-99) determined by the specific states and stack symbols defined for two digits.
- **Scalability:** Extending this specific design to three or more digits would require adding more stack symbols (e.g., $H_0..H_9$ for hundreds) and potentially more intermediate carry states (e.g., $q_{\text{inc\_hundreds}}$), increasing the complexity of the PDA definition.
- **Alternative Models:** While this model works, the challenges encountered when trying to implement the original *recursive* infinite counter highlight that certain intuitive algorithms may require more sophisticated automaton techniques (like multi-tape or multi-stack machines, or different state/stack encodings) or might naturally fit other computational models better.
- **Output Mechanism:** As with the conceptual recursive model, reading the final count requires interpreting the stack configuration $(\#, T_X, U_Y)$ to produce the numerical value $10X + Y$.

11 Conclusion

This exercise demonstrates that a Deterministic Pushdown Automaton can successfully model base-10 counting within a fixed range (0-99) by carefully managing state transitions and using distinct stack symbols to represent place values. The carry operation, which proved difficult in a purely recursive, unbounded model using simple stack symbols, is handled effectively here through an intermediate state ('q_{inc_tens}') that processes the carry after the triggering digit has been popped.

The successful implementation of the two-digit counter contrasts with the difficulties faced when attempting a direct implementation of the unbounded recursive counter described initially. This suggests that while PDAs are more powerful than FSAs due to their stack memory, the specific constraints on stack access (top symbol only) require careful design choices when modeling algorithms that seem to require looking deeper into the data structure, such as multi-digit arithmetic carries.

11.1 Key Takeaways

- **Place Value Simulation:** PDAs can represent place values using distinct stack symbols and structure (e.g., $(\#, \text{Tens}, \text{Units})$).
- **State-Managed Carry:** Carry operations between fixed place values can be implemented using dedicated states and epsilon transitions that manipulate the stack after the relevant digit is exposed.
- **Finite vs. Infinite:** Modeling fixed-range counting is feasible with standard PDAs, while unbounded counting presents significant design challenges for the standard single-stack PDA model if aiming for a simple recursive digit representation.
- **Model Suitability:** The choice of automaton model (FSA, PDA, Turing Machine, etc.) should consider the specific memory and access patterns required by the algorithm being modeled.

Python Test

```
# Import necessary classes from automata-lib
try:
    from automata.pda.dpda import DPDA
    from automata.pda.stack import PDAStack
    from automata.base.exceptions import RejectionException
except ImportError:
    print("Error: automata-lib not found.")
    print("Please install it: pip install automata-lib")
    # Mocking classes if needed
    class MockPDAConfiguration:
        def __init__(self, state, stack_tuple): self.state, self.stack = state, self._MockStack(stack_tuple)
        class _MockStack:
            def __init__(self, stack_tuple): self.stack = stack_tuple
    class MockDPDA:
        def __init__(self, *args, **kwargs): self.final_states = kwargs.get('final_states', set()); print("Warning: Using Mock DPDA class.")
        def read_input(self, input_sequence):
            n = len(input_sequence)
```

```
            if n > 99: return MockPDAConfiguration('q_halt', ('#', 'T0', 'U0')) # Mock halt
            if n == 0: return MockPDAConfiguration('q_idle', ('#', 'T0', 'U0'))
            tens, units = divmod(n, 10)
            stack_list = ('#', f'T{tens}', f'U{units}')
            return MockPDAConfiguration('q_idle', tuple(stack_list))
    DPDA = MockDPDA
    RejectionException = Exception

import sys

# Define the components of the 2-Digit Counter PDA

# States
states = {'q_start', 'q_idle', 'q_inc_tens', 'q_halt'}

# Input symbols
input_symbols = {'tick'}

# Stack symbols
stack_symbols = {'#'} | {f'T{i}' for i in range(10)} | {f'U{i}' for i in range(10)}

# --- CORRECTED Transitions ---
transitions = {
    'q_start': {
        '': {
            # Initial: Pop '#', Push '#', Push 'T0', Push 'U0'. Stack ('#', 'T0', 'U0'). Top 'U0'.
            '#': ('q_idle', ('U0', 'T0', '#'))
        }
    },
    'q_idle': { # Normal state, processing 'tick' on Units digit
        'tick': {
            # Inc Units < 9: Pop Un, Push U(n+1). Stack (..., Tx, U(n+1)). Stay q_idle.
            **{f'U{n}': ('q_idle', (f'U{n+1}',)) for n in range(9)},

            # Inc Units = 9: Pop U9, Push nothing. Stack (..., Tx). Go to q_inc_tens state to process Tx.
            'U9': ('q_inc_tens', ()) # CORRECTED: Don't push U0 here
        }
    },
    'q_inc_tens': { # Epsilon transitions to handle the Tens digit (which is now on top)
        '': {
             # Tens digit Tm (m<9): Pop Tm. Push T(m+1), then Push U0. Stack (..., T(m+1), U0). Return to q_idle.
             **{f'T{m}': ('q_idle', ('U0', f'T{m+1}')) for m in range(9)}, # CORRECTED: Push U0 here

            # Tens digit T9 (Overflow): Pop T9. Push T0, then Push U0. Go to q_halt state.
            'T9': ('q_halt', ('U0', 'T0')) # CORRECTED: Push U0 here
        }
    },
    'q_halt': {
```

```
        # No transitions out of halt state
    }
}

# Initial state
initial_state = 'q_start'
initial_stack_symbol = '#'
final_states = {'q_idle'} # Only q_idle is accepting

# Create the DPDA instance
try:
    pda = DPDA(
        states=states,
        input_symbols=input_symbols,
        stack_symbols=stack_symbols,
        transitions=transitions,
        initial_state=initial_state,
        initial_stack_symbol=initial_stack_symbol,
        final_states=final_states,
        acceptance_mode='final_state'
    )
except Exception as e:
     print(f"Error creating DPDA: {e}")
     # Setup Mock DPDA if needed
     class MockPDAConfiguration:
        def __init__(self, state, stack_tuple): self.state, self.stack = state, self._MockStack(stack_tuple)
        class _MockStack:
             def __init__(self, stack_tuple): self.stack = stack_tuple
     class MockDPDA:
        def __init__(self, *args, **kwargs): self.final_states = kwargs.get('final_states', set()); print("Warning: Using Mock DPDA class after creation error.")
        def read_input(self, input_sequence):
             n = len(input_sequence)
             if n > 99: return MockPDAConfiguration('q_halt', ('#', 'T0', 'U0'))
             if n == 0: return MockPDAConfiguration('q_idle', ('#', 'T0', 'U0'))
             tens, units = divmod(n, 10)
             stack_list = ('#', f'T{tens}', f'U{units}')
             return MockPDAConfiguration('q_idle', tuple(stack_list))
     pda = MockDPDA(final_states=final_states)
     RejectionException = Exception
     print("--- Proceeding with Mock PDA ---")


# Function to convert the stack contents to an integer (Unchanged)
def stack_to_int_2digit(stack_tuple: tuple) -> int:
    if not (isinstance(stack_tuple, tuple) and len(stack_tuple) == 3 and \
            stack_tuple[0] == '#' and stack_tuple[1].startswith('T') and \
            stack_tuple[2].startswith('U')):
        print(f"Warning: Invalid stack state for 2-digit conversion: {stack_tuple}")
        return -1
    try:
        tens_digit = int(stack_tuple[1][1:])
        units_digit = int(stack_tuple[2][1:])
```

```
        return tens_digit * 10 + units_digit
    except (ValueError, IndexError):
        print(f"Error converting stack digits to int: {stack_tuple}")
        return -2

# --- Testing the PDA ---
print("Testing 2-Digit (0-99) Counter PDA:")
test_counts = [0, 1, 9, 10, 11, 19, 20, 98, 99, 100, 101]

for count in test_counts:
    print(f"\n--- Testing count = {count} ---")
    input_sequence = ['tick'] * count
    try:
        final_config = pda.read_input(input_sequence)
        final_state = final_config.state
        if hasattr(final_config, 'stack') and hasattr(final_config.stack, 'stack'):
             final_stack_tuple = final_config.stack.stack
        else:
             print("Error: Final configuration object has unexpected structure.")
             final_stack_tuple = ('#', 'ERROR', 'ERROR')

        is_accepted = final_state in pda.final_states

        print(f"Input: {count} 'tick's")
        print(f"Ended in State: {final_state}")
        print(f"Final Stack: {final_stack_tuple}")

        expected_value = count if count <= 99 else -1
        expected_acceptance = (count <= 99)

        print(f"Expected Acceptance: {expected_acceptance}")
        print(f"Actual Acceptance: {is_accepted}")

        if is_accepted:
            calculated_value = stack_to_int_2digit(final_stack_tuple)
            print(f"Expected Value (if accepted): {expected_value}")
            print(f"Calculated Value: {calculated_value}")
            if calculated_value == expected_value and expected_acceptance:
                print("Result: Correct")
            else:
                print("Result: INCORRECT (Value mismatch or unexpected acceptance)")
        else: # Rejected (ended in non-final state like q_halt)
            print("Expected Value (if accepted): N/A")
            print("Calculated Value: N/A (Rejected)")
            if not expected_acceptance:
                 print("Result: Correct (Rejected as expected)")
            else:
                 print("Result: INCORRECT (Unexpected rejection)")

    except RejectionException as re:
        # This indicates the PDA got truly stuck (no transition defined)
        print(f"Input: {count} 'tick's")
        print(f"PDA Rejection Exception: {re}")
        print("Result: REJECTED (Stuck) - Check Logic")
```

```
172
173     except Exception as e:
174         print(f"Input: {count} 'tick's")
175         print(f"PDA Error: {e}")
176         # import traceback # Uncomment for debugging
177         # traceback.print_exc() # Uncomment for debugging
178         print("Result: ERROR")
```