Report

Name: Vishakha Agrawal

Roll Number: 2023101040

1. Pseudo Random Number Generator (PRNG)

Data Structures

machine: Dictionary where the key is the current state, and the value is the next state. This structure stores the DFA transitions.

frequency: Dictionary that keeps track of how many times each state is visited. The keys are state numbers (1 to N), values are the visit counts

visit: A dictionary used to store the step at which each state is first visited. This helps in detecting cycles.

cycle: It is a list that stores the order of states in the cycle.

Functions:

initialize_frequency(N):

We initialize the frequency of all states to be 0.

def build_machine():

Read input and construct the machine dictionary. Call the cycle detection function to compute the frequencies of the states and print them.

run_dfa(N, S, X, machine):

Logic:

We start at the starting state S. We read the complete input from the input tape. We loop through all the states in the machine which have outgoing transitions. If the current_state is in the machine, there is a valid transition to the next state. We fetch the next state from the machine and update the current state to the next state.

The frequency of a previous transition is only incremented ***after*** transition takes place. Incrementing it before the transition takes place will cause errors in cases where the DFA is sparse.
Example test case taken to confirm this:
1
100000 1 1 1000000000000

1 100000

Expected output:

1 0 0 . . . 0

Output when frequency is updated before transition:

1 0 0 0 . . . 1

cycle_detection(N, S, X, machine):

This function is critical as it optimizes the code for cycles. Without this the run will not terminate in a reasonable time in large cases. It avoids redundant computation if cycles are detected.

We define a visit dictionary to keep track of the first time each state was visited in the run. We also keep track of which step the state was visited at. If the run visits a previously marked state, then we know at which step and state a cycle begins.

Cycle Detection:

This loop runs if the current state hasn't been visited before and there is a valid transition from the current state. The sate and the step at which it is first encountered is stored in the visit dictionary. A list stores the states in the potential cycle.

No cycle: If the DFA reaches a state that is still not visited, the DFA is simulated for all X steps.

Cycle detected: If a state is visited again in some step, a cycle is detected starting from the current state. The length obtained by subtracting is the step number when the cycle started from the current step.

We simulate the DFA until it reaches the start of the cycle normally. Once the DFA enters the cycle, we calculate how many full cycles fit into the remaining steps. After completing the full cycles, we simulate any remaining steps that don't complete another full cycle.

The final frequency is calculated by combining the pre-cycle steps, the full cycles (scaled by the number of times the cycle is repeated), the remaining steps after the full cycles.

2. L-Systems

Simulating L-systems

1. "I'm a mirrorball"

We begin by adjusting the canvas and scaling it.

We define an Lsystem class and define the starting string of the L-system (axiom). We then define the productions rules and finals. The finals object maps each symbol to a specific drawing action.

If the symbol is F, ctx.beginPath() draw a vertical line from the current position to a new position 25 / (ball.iterations + 1) units down. The length of the line **decreases** as the number of iterations increases. ctx.translate() updates the position of the pen.

If the symbol is G, we move forward the same distance without drawing anything. Similarly, we implement all the other symbols.

The rules are applied for a specific number of iterations and the user defined getString() function retrieves the final string.

ball.final() processes the generated string and executes the drawing commands.

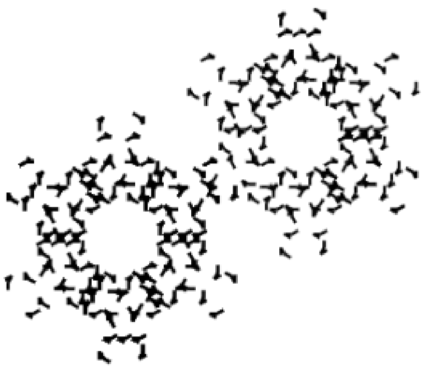


Figure 1: Mirrorball (9 iterations)

2. "Is that a tree?"

Since my roll number ends in 040, angle of rotation for tree = 10-15+10*(-1)^(5)=-5-10=-15

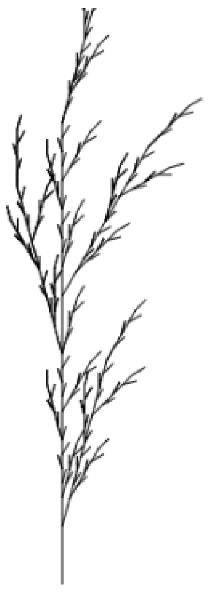


Figure 2: Tree (4 iterations)

3. "Anything that can happen will happen"

When the L-system encounters the symbol F, this function is invoked, which selects one of three possible expansion options :

'F[F]F'

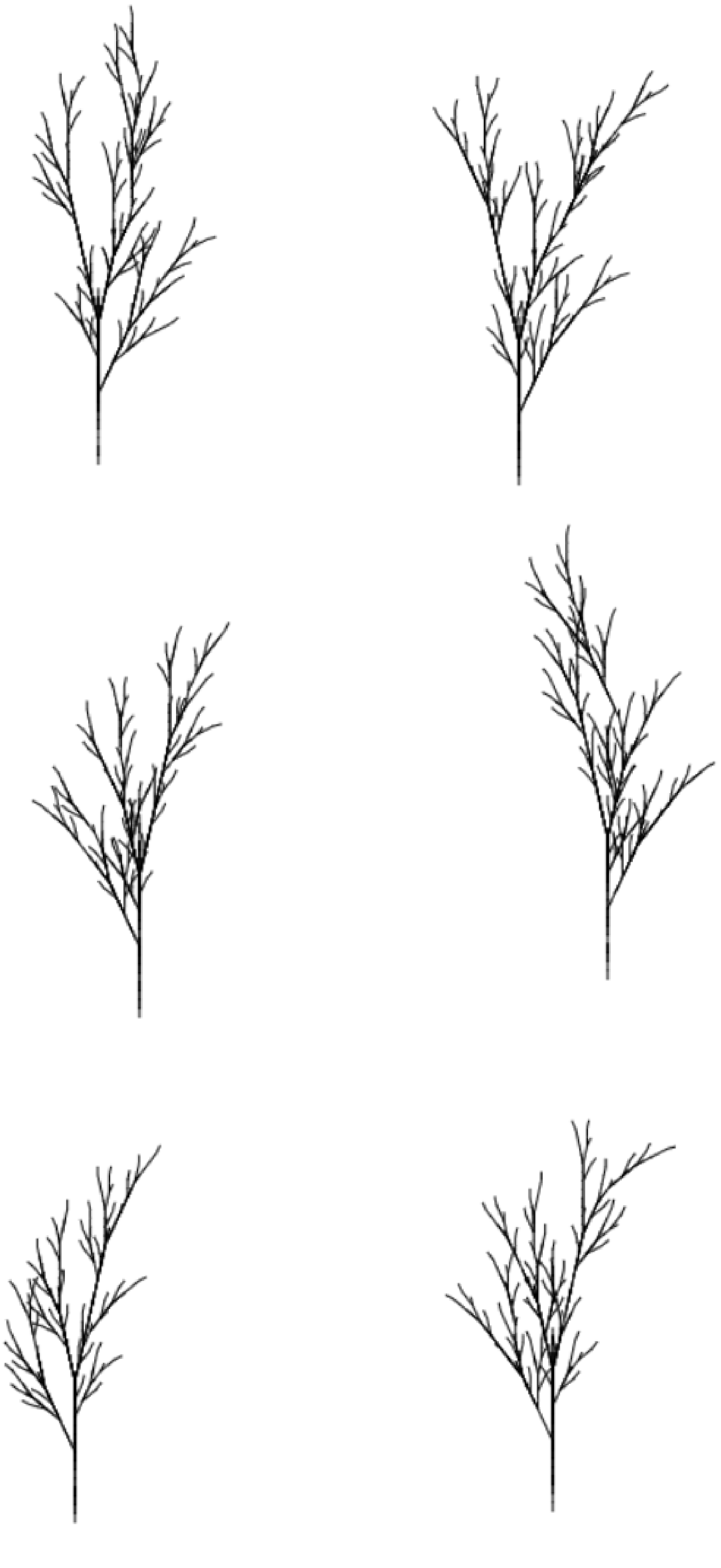
'F[+]F'

'F[FF]F'

The function randomly picks one of these three options using:

Math.floor(Math.random() * options.length)

The same applies in the case of X.



4. “Content without context is noise"

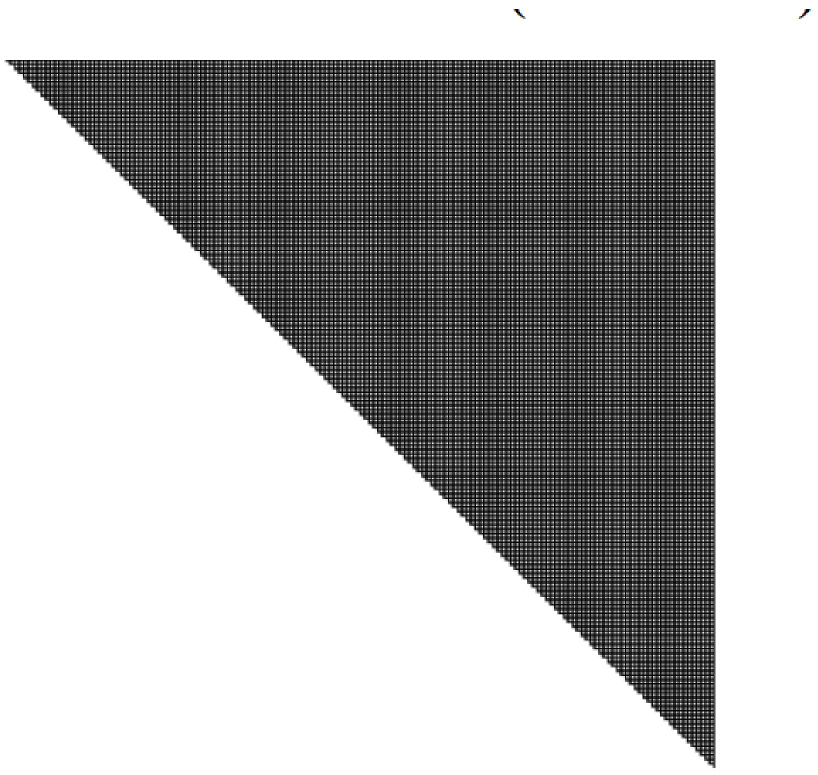


Figure 3: Context (9 iterations)

context_sensitive_rule(productions, input):
This function iterates over each character in the input string. If the context for a particular character matches both the left and right context as in the rule, the character is replaced with F-+F+F. Else, it applies the normal production rule for the current character. If no rule matches, the current character is not changed.

iterate_LSystem(axiom, productions, iterations):

Applies the rules iteratively and returns the final string.

Expanding the axiom:

F+F

F - F + + F - F + F - F + + F – F

As we keep applying the rules, we do not obtain a pattern of the form FF-F to apply the context-sensitive rule.

2. Find the axioms

1. Stick Tree

axiom: F

productions:

F → X[+F] F[-F] YF

X → XX

Y → YY

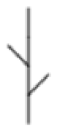
Both F and X are used to draw

Intuition

There could not be only one rule or any perfectly symmetric rule because that would lead to a perfectly symmetric propagation of the tree. The starting branch or root in i = 2 is twice as long as the branches, therefore X -> XX is added. On expanding the grammar for i =2, we must be in a situation where on restoring the branch we are able to restore context to the original root. Therefore, we introduce a variable F the produces branches and restores context on completion. We have the Y -> YY due to the length of the progressing tree observed in the iteration 2. The completely symmetric grammars were generating branches in the segment where there were none.

Iteration 0: F

Iteration 1: X[+F]F[-F]YF



Iteration 2: XX[+X[+F]F[-F]YF]X[+F]F[-F]YF[-X[+F]F[-F]YF]YYX[+F]F[-F]YF

2. Koch Santa

axiom: F

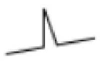
productions:

F → F+F--F+F

Intuition:

In the beginning, we would want to have 4 line-segments such that two of them form a sharp peak between the other two. The - - in the rule is to this effect. It causes this sharp 160-degree turn. This reverses the direction of the line and creates a sharp change in the curve's direction. The F- -F is flanked by two +Fs to retain the original line. The double left turn (--) gives the Koch curve its zigzag pattern. Unlike the traditional 60-degree angle of a Koch curve, a steeper angle was chosen to make the figure more angular. In every iteration, each line segment is replaced by four smaller segments in the shape of a peak. This is done recursively.

Iteration 1:
F+F--F+F



Explanation:
First we move forward, then we take a right turn by 80 degrees. We move forward and take a sharp 160-degree turn. Then we move forward again and turn right to realign the curve.

Iteration 2:

F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F