Report: 2023113019:

Q1.First and foremost, we model this particular DFA in the form of a graph(of sorts), with edges and number of nodes being user input.

The approach used thereby is by getting a cycle detected. This is due to the observation that there cannot be X > number of states so long as there are no repetitions(by pigeon-hole principle). Hence, there must exist a cycle if X > number of states. If not, it can still have a cycle, or can terminate.

In either case, we notice that a simple array can simulate both the solution, and the cycle detection, because a cycle just means that the element comes again while running through the input. So what we do is use the transitions( simulated as edges here) to run through the graph (iterate till X is reached or cycle is detected) while updating their solution values(also stored in an array). As soon as an element is recognised the second time, we understand that it must belong to a cycle, and we add it to the cycle array. As soon as an element is recognised for the third time, we understand that the cycle has started repeating, so we terminate the simulation.

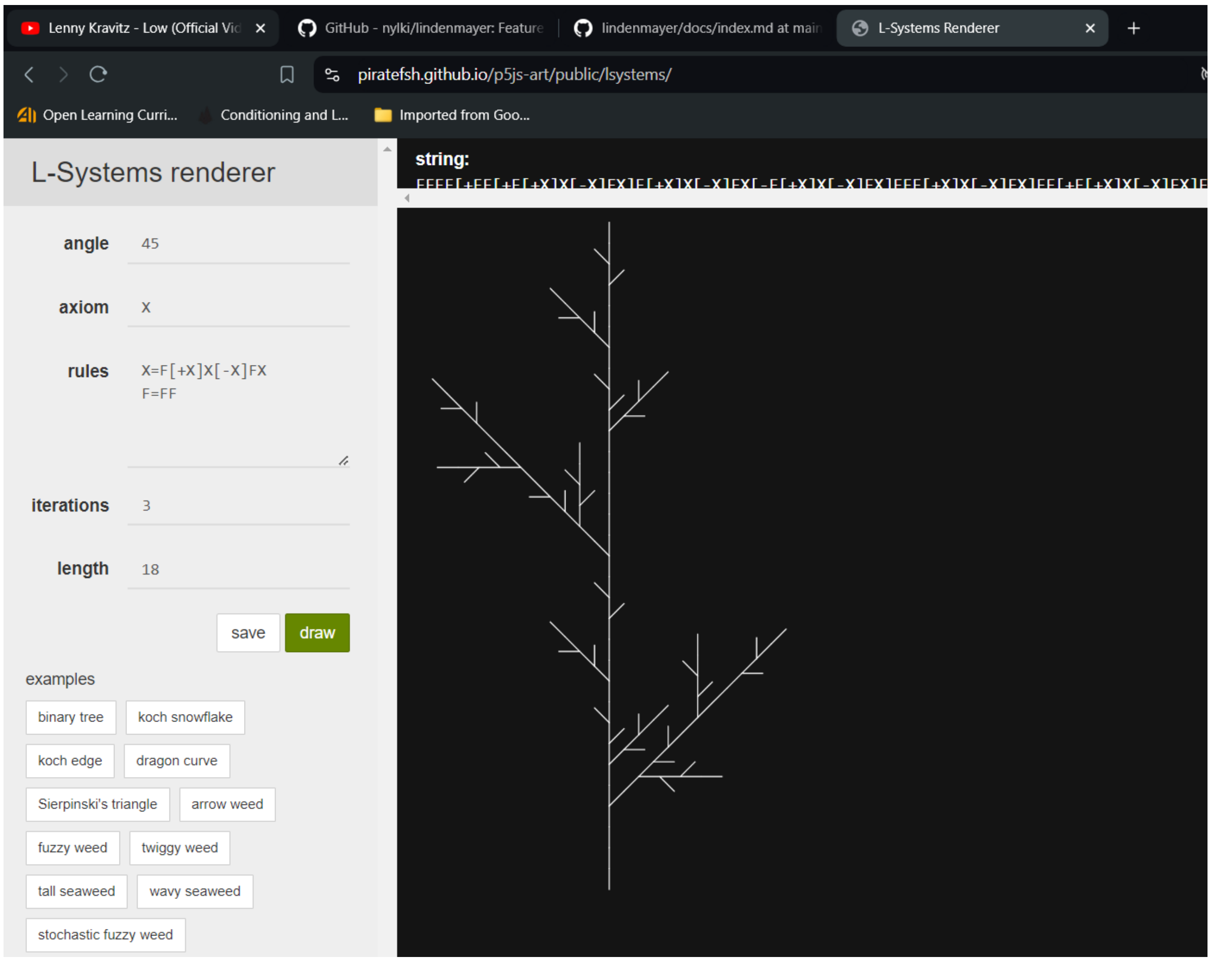
All of this has been done so that we can directly calculate solution of cycle iterations by using x-count(of the iterations) floor divided by the length of cycle and add it to the solutions of each cycle. The remaining values of X(less than length of cycle) are thereby run again. This ensures a time complexity of O(number of states[length of cycle cannot be greater than that, and if no cycle, it cannot run for more than that(pigeonhole)])

Q2. All images have been attached in the q2 subfolder. They have been implemented using the Javascript library which was given to us as a reference, with slight modifications to the axioms, production rules, and final states. We notice that [, and ] need to be added as ctx.save() and ctx.restore(), and are not given by default.
Roll number is 2023113019: r = 19, and that is the basis of the calculation.

The randomness has been implemented using the documentation.

Q3. https://piratefsh.github.io/p5js-art/public/lsystems/ Was used to visualise the results.

The intuition behind the stick-tree question was just about noticing where the iterations cause repeating units. I noticed that all those must be a variable and not a terminal. Also noticed that they were primarily just getting a right and left terminal with the X remaining.


Lenny Kravitz - Low (Official Vid
GitHub - nylki/lindenmayer: Feature
lindenmayer/docs/index.md at main
L-Systems Renderer
piratefsh.github.io/p5js-art/public/lsystems/
Open Learning Curri...
Conditioning and L...
Imported from Goo...

L-Systems renderer

angle 45

axiom X

rules

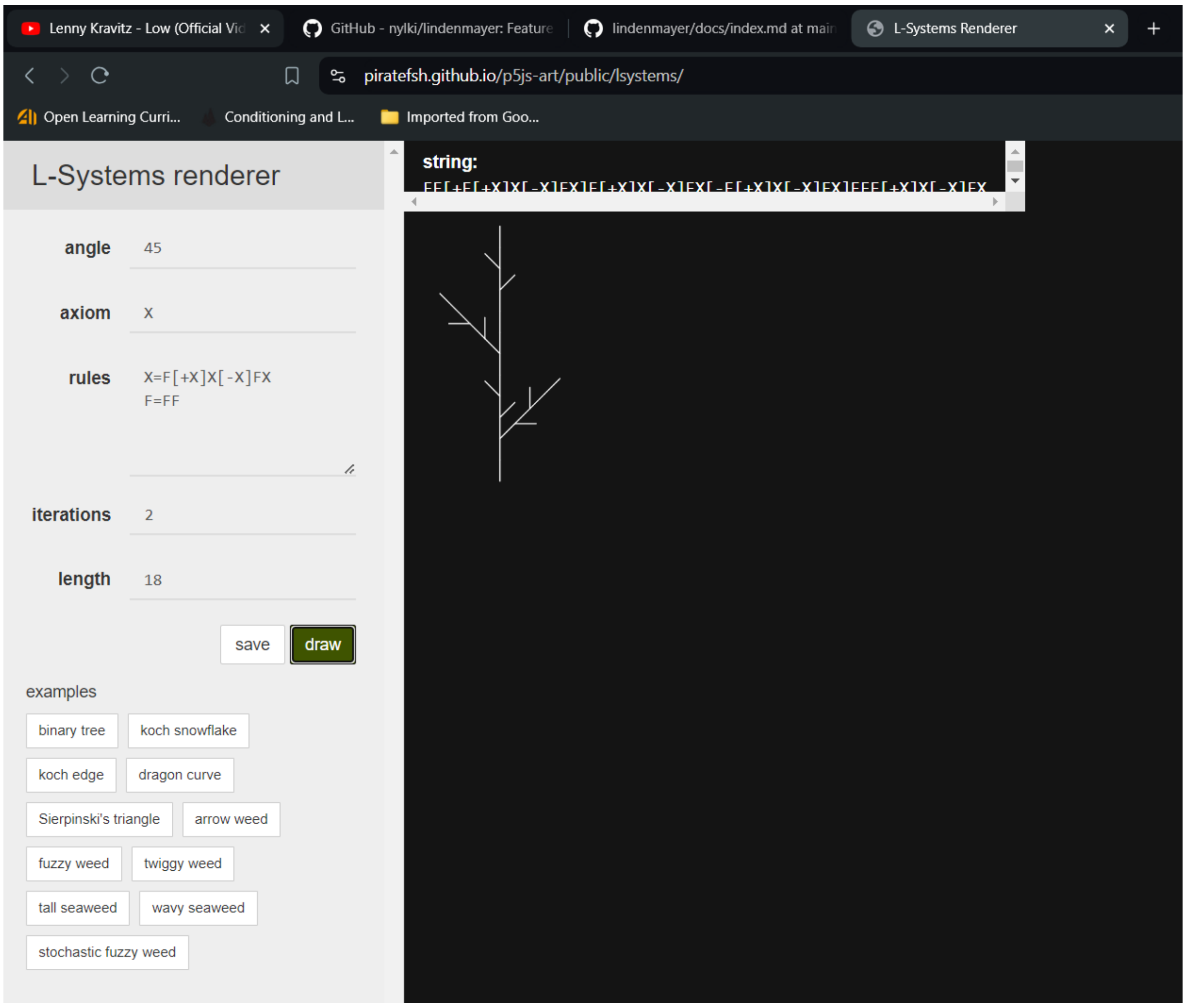
```
X=F[+X]X[-X]FX
F=FF
```

iterations 3

length 18

string:

```
FFFF[+FF[+F[+X]X[-X]FX]F[+X]X[-X]FX[-F[+X]X[-X]FX]FFF[+X]X[-X]FX]FF[+F[+X]X[-X]FX]F
```



piratefsh.github.io/p5js-art/public/lsystems/

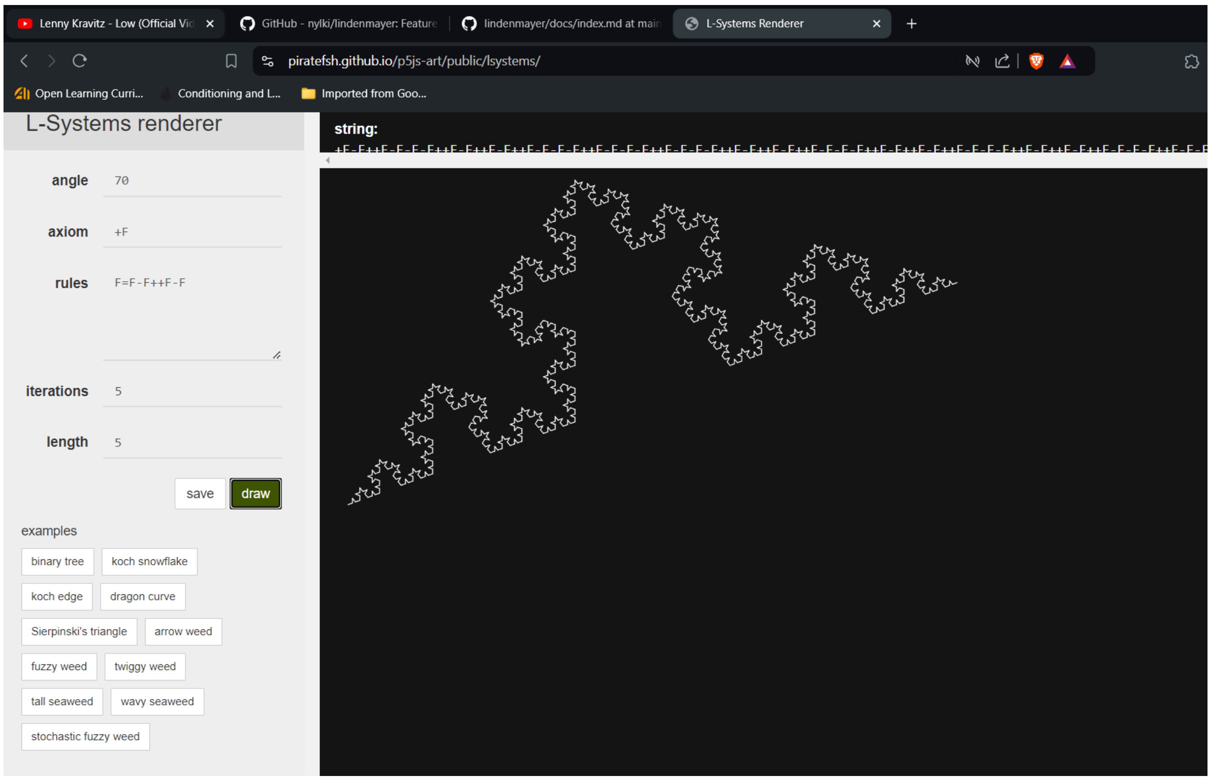
L-Systems renderer

| | |
|---|---|
| angle | 45 |
| axiom | X |
| rules | X=F[+X]X[-X]FX<br>F=FF |
| iterations | 2 |
| length | 18 |

string:

```
FF[+F[+X]X[-X]FX]F[+X]X[-X]FX[-F[+X]X[-X]FX]FFF[+X]X[-X]FX
```

Santa: The clue is to observe the repeating unit of  a straight line to -/\-, and we could achieve that just using F, and transforming it to the rule written below.



L-Systems renderer

angle 70

axiom +F

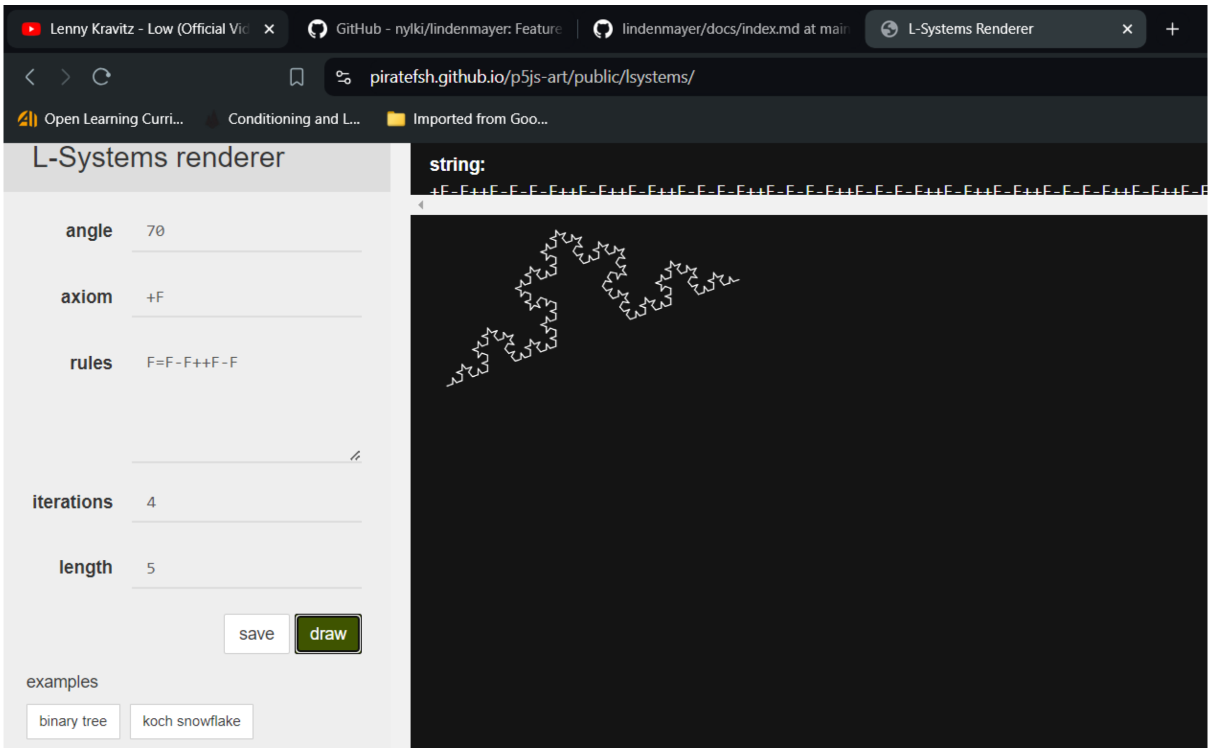
rules F=F-F++F-F

iterations 5

length 5

string:

```
+F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F
```



L-Systems renderer

angle 70

axiom +F

rules F=F-F++F-F

iterations 4

length 5

string:

```
+F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F-F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F
```

Lenny Kravitz - Low (Official Vic
GitHub - nylki/lindenmayer: Feature
lindenmayer/docs/index.md at main
L-Systems Renderer
piratefsh.github.io/p5js-art/public/lsystems/
Open Learning Curri...
Conditioning and L...
Imported from Goo...
save
draw
examples
binary tree
koch snowflake