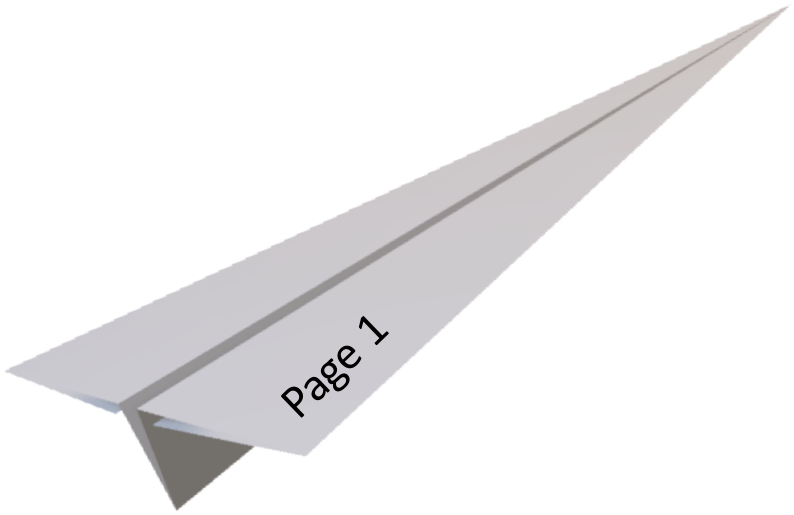
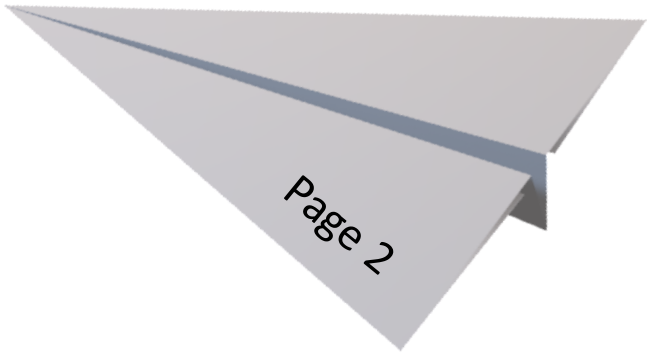
Redesigning Sipser's NFA into a One-Cycle Form



Language, Computation, and Machines Presentation
By Sahil Chhokar and Karan Dhanoa

COMP-382-ON1
Group 7

Introduction



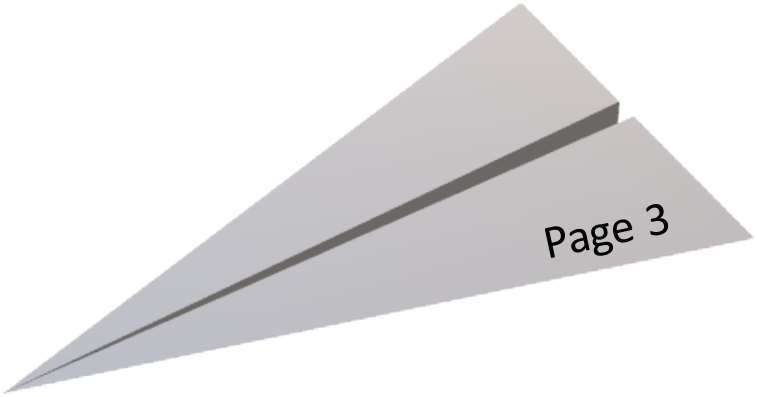
An NFA, or Nondeterministic Finite Automaton, is a type of state machine in automata theory. It reads a string of symbols step by step, then moves through states based on rules. The “nondeterministic” part means the machine can sometimes make choices, like jumping into one path or another, and as long as one path leads to acceptance, the string is accepted.

The specific NFA we’re looking at comes from Michael Sipser’s textbook. It accepts strings made up only of the number 0, where the length of the string is divisible by 2 or by 3. For example, “00” (length 2) is accepted, “000” (length 3) is accepted, “000000” (length 6) is accepted, so on and so forth.

The original design works to accomplish this task, but it uses an epsilon transition. This means that it essentially has a free move where the automaton doesn’t read any input symbol. That free move is used to “guess” whether it should take the cycle for multiples of 2 or the cycle for multiples of 3. This makes the design a little messy and repetitive.

Our goal is to redesign this machine to be cleaner. Instead of two separate loops, we’ll build a single six-state cycle that handles both conditions at once. To do that, we’ll use modular arithmetic, which means, “remainders when you divide numbers.”

In this presentation, we review the following:



1. Review Sipser's original nondeterministic automaton and how it works.
2. Introduce the redesigned one-cycle automaton using modular arithmetic.
3. Demonstrate the equivalence with a Python code test and table of results.
4. Build an empirical explanation using remainders modulo 6.
5. Present a full formal proof step by step.
6. Wrap up with key takeaways about NFAs, nondeterminism, and modular arithmetic.

Original NFA

The original automaton is called N3 in Sipser's book. Here's how it works:

- The machine starts at the beginning and immediately takes an epsilon transition. That means it splits into two possible paths without reading anything.
- On the first path, the machine enters a two-state cycle. This loop accepts strings of even length, because after every 2 steps it lands back in the accepting state.
- On the second path, the machine enters a three-state cycle. This loop accepts strings whose lengths are multiples of 3, because after every 3 steps it lands back in an accepting state.

So, for example:

- If the machine takes "0000" (length 4), the two-state path would accept it, therefore it is accepted, even if the 3-state cycle rejects.
- If the machine takes "00000" (length 5), neither path accepts it.
- If the machine takes "000000" (length 6), both paths would accept it, because 6 is divisible by 2 and 3.

This design is correct, but it uses nondeterminism. The epsilon move at the start is essentially guessing the divisibility condition, but we can do better by removing that guesswork.

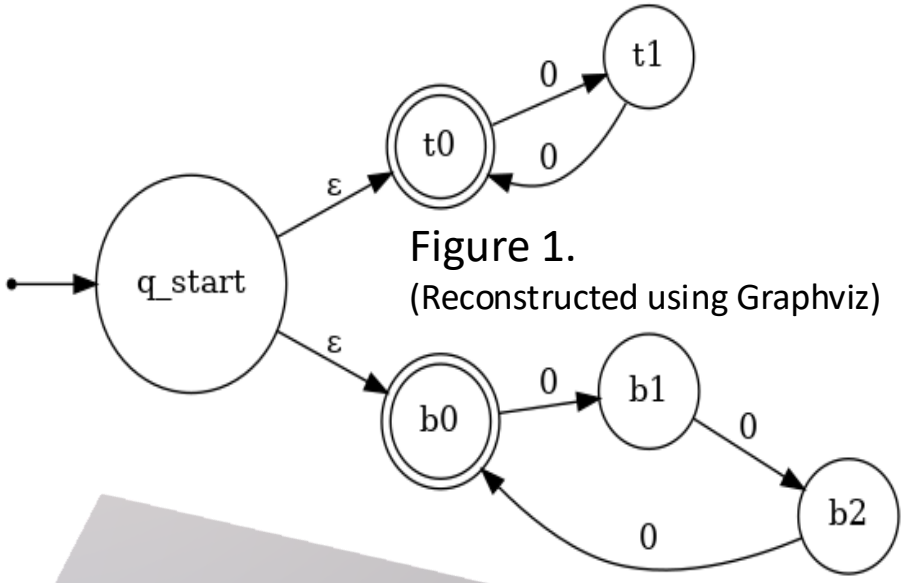


Figure 1.
(Reconstructed using Graphviz)

Redesigned DFA

The redesigned automaton, which is now a DFA since it no longer has a “choice” in the path, avoids the epsilon transition and instead uses one single loop of six states.

We use six states, since six is the least common multiple of 2 and 3. That means divisibility by 2 and by 3 can both be accepted or rejected when looking at the remainders when dividing by 6.

Each state represents the remainder when dividing the string’s length by 6.
For example, after reading 7 zeros, the machine is in state q1, because 7 divided by 6 leaves a remainder of 1.

The accepting states are chosen with respect to the remainder and divisibility by 2 or 3. Those remainders are 0, 2, 3
Remainder 0: divisible by both 2 and 3.
Remainder 2 or 4: divisible by 2.
Remainder 3: divisible by 3.

This way, the machine handles both divisibility conditions using a single cycle of six states. It’s more compact and removes the nondeterministic choice at the start.

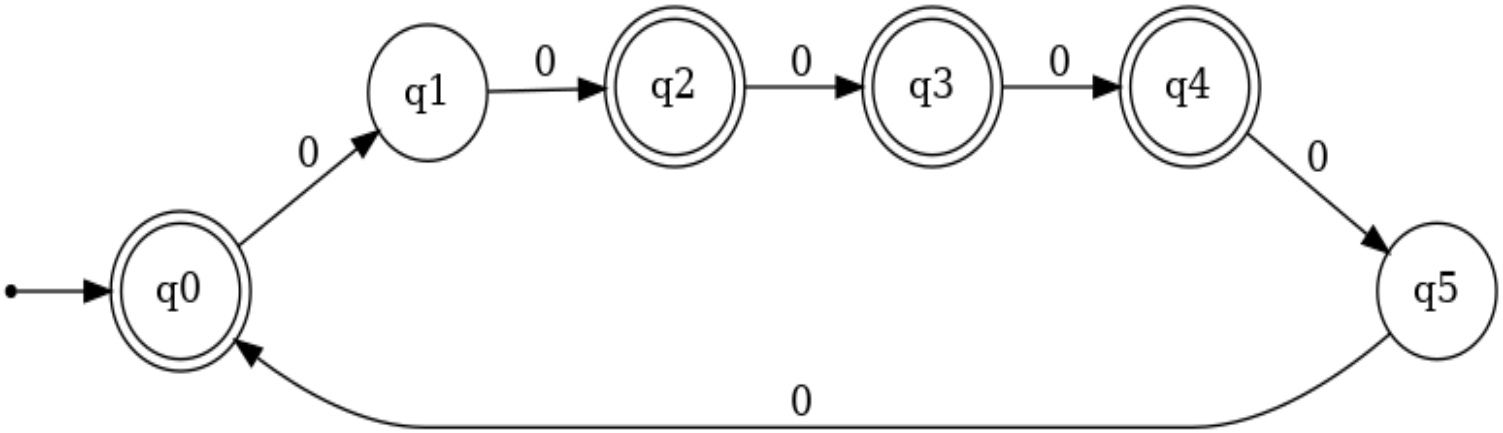
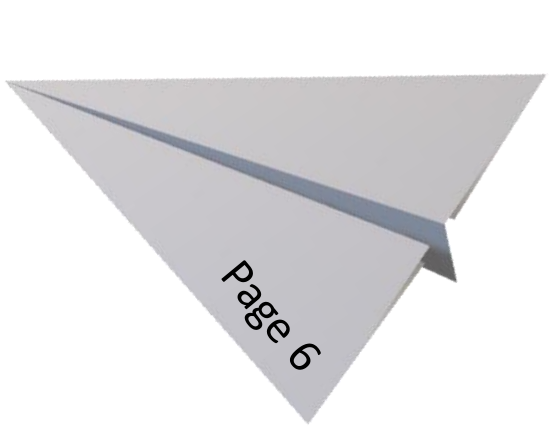


Figure 2.

Code Demo



To check that this redesign accepts the same language as the original NFA, we have constructed an example Python file to visually show the equivalence before a more formal proof.

The program defines two functions:

- The first checks divisibility by using the logic of the original NFA, that is, seeing if the string length is divisible by 2 or 3.
- The second checks divisibility using the remainder logic of the redesigned DFA. That is, checking if the remainder is 0, 2, 3, or 4, when dividing the string length by 6.

We then run both functions on string lengths from 0 through 20 and print the results side by side. The output shows that both machines always match, whenever the original accepts, the redesigned accepts too, and whenever one rejects, the other rejects as well.

We technically only need to test 0 through 5, because the remainders repeat every 6 numbers. But running the test up to 20 makes the repeating pattern more obvious to see.

Assignment1 NFA.py

Users > sahilchhokar > Documents > Assignment1 NFA.py > ...

```python
# Original NFA acceptance: divisible by 2 or 3
def original_accepts(k):
    return (k % 2 == 0) or (k % 3 == 0)

# Redesigned NFA acceptance: divisible by 2 or 3 via mod 6 residues
def redesigned_accepts(k):
    return (k % 6) in {0, 2, 3, 4}

# Compare results for 0 to 20
print("Length | Original | Redesigned")
for k in range(21):
    print(f"{k:6} | {original_accepts(k)}     | {redesigned_accepts(k)}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
sahilchhokar@Mac ~ % /usr/local/bin/python3 "/Users/sahilchhokar/Documents/Assignment1 NFA.py"
Length | Original | Redesigned
     0 | True     | True
     1 | False    | False
     2 | True     | True
     3 | True     | True
     4 | True     | True
     5 | False    | False
     6 | True     | True
     7 | False    | False
     8 | True     | True
     9 | True     | True
    10 | True     | True
    11 | False    | False
    12 | True     | True
    13 | False    | False
    14 | True     | True
    15 | True     | True
    16 | True     | True
    17 | False    | False
    18 | True     | True
    19 | False    | False
    20 | True     | True
sahilchhokar@Mac ~ %
```

Empirical Comparison

Residue (mod 6)	Original NFA	Redesigned DFA
0	Accept	Accept
1	Reject	Reject
2	Accept	Accept
3	Accept	Accept
4	Accept	Accept
5	Reject	Reject

The table here compares all six remainder classes: 0, 1, 2, 3, 4, and 5. No matter how long the string is, it always falls into one of these six categories when divided by 6.

To further explain this, if you take any number and divide by 6, the remainder is always between 0 and 5. For example:

- 8 divided by 6 is 1 remainder 2.
- 14 divided by 6 is 2 remainder 2.
- 20 divided by 6 is 3 remainder 2.

Even though 8, 14, and 20 are varyinglengths, they're all in the same "remainder 2" class.
If we mark which remainders correspond to divisibility by 2 or 3, we get:

- 0: divisible by 2 and 3 = accept
- 2: divisible by 2 = accept
- 3: divisible by 3 = accept
- 4: divisible by 2 = accept
- 1 and 5: divisible by neither = reject

This shows why we only need six states in the new automaton/ DFA. By covering these six remainder cases, we've covered every possible input length. The table demonstrates this, both the original and redesigned NFAs behave the same.

Formal Proof Outline

Now let's outline the proof that the redesigned automaton is equivalent to the original.

Step 1: State tracking. We need to show that after reading k zeros, the redesigned automaton is in state q(k mod 6). In plain English, this states: "the machine really does keep track of the remainder when dividing the length by 6."

Step 2: Acceptance condition. Next, we need to show that the machine accepts if and only if the state is one of {q0, q2, q3, q4}. This is because those states represent exactly the remainders that correspond to divisibility by 2 or 3, of which those remainders are 0, 2, 3, and 4, respectively.

Step 3: Arithmetic connection. Finally, we connect it all together. We show that if a number is divisible by 2 or 3, then its remainder modulo 6 is in {0, 2, 3, 4}, and if its remainder is in that set, then the number is divisible by 2 or 3. This is the most important step because it matches the new machine's accepting states to the original machine's condition.

By completing these three steps, we prove that the redesigned NFA and the original NFA accept the same language.

Step 1 (State Tracking): We prove by induction. For the base case (k = 0), the machine is at q0, which matches 0 mod 6 = 0. For the inductive step, if the claim is true for k, then after one more zero it moves to q((k+1) mod 6), so the claim is true for all k.

Step 2 (Acceptance Condition): The definition of the machine says that {q0, q2, q3, q4} are accepting states. So the machine accepts if and only if the current state is in that set.

Step 3 (Arithmetic Connection):

- If k is divisible by 2, its remainder mod 6 must be 0, 2, or 4.
- If k is divisible by 3, its remainder mod 6 must be 0 or 3.
- Together, these give the set {0, 2, 3, 4}.
- Conversely, if the remainder is in {0, 2, 3, 4}, then k is divisible by 2 or 3.

This proves the two machines recognize the same language.

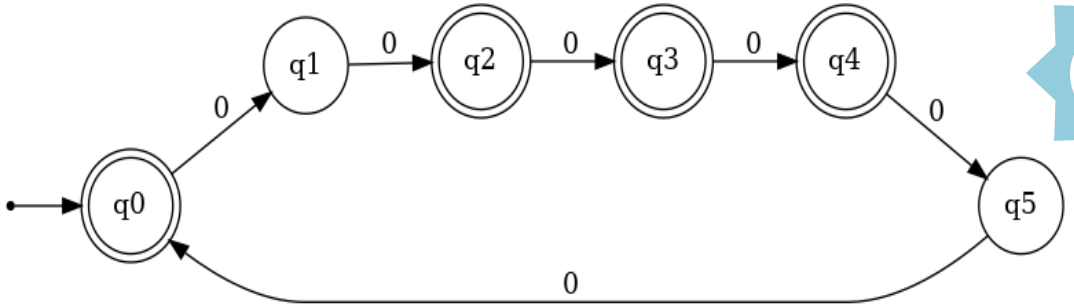


Figure 2 for reference

Now, we put it all together in one formal statement.

We define the redesigned automaton M as:

- States: {q0, q1, q2, q3, q4, q5}
- Alphabet: {0}
- Start state: q0
- Accepting states: {q0, q2, q3, q4}
- Transition: δ(qi, 0) = q(i+1 mod 6)

Note: The above "transition", simply means for the LHS with the Greek symbol δ, that if it's in state X and if is sees symbol y, where does it go next. On the RHS, this simply states that if it's in state qi and if it reads another 0, then move forward to the next state around the 6-state cycle.

Claim: After reading k symbols, the machine is in state q(k mod 6).

We prove this by induction:

- Base case: k = 0 → still at q0, matches remainder 0.
- Inductive step: If true for k, then one more zero leads to q((k+1) mod 6).

Therefore, the claim is true for all k. The machine accepts if and only if the current state is in {q0, q2, q3, q4}, which happens exactly when the length k is divisible by 2 or 3.

Thus, the redesigned automaton recognizes the same language as the original.

Key Takeaways

We started with Sipser's original NFA, which used nondeterminism and epsilon transitions to handle divisibility by 2 and 3 separately. Then, by thinking in terms of remainders and modular arithmetic, we redesigned it into a simpler six-state cycle.

We backed up this redesign in three ways:

1. A code demonstration that tested lengths up to 20 and showed both machines always agreed.
2. A remainder table that proved we only need to look at remainders modulo 6 to cover all cases.
3. A detailed formal proof that connected the automaton's states to divisibility conditions.

All of these confirm the same result, that the redesigned NFA accepts exactly the same language as the original. More importantly, this example shows how we can sometimes remove nondeterminism and still capture the same behavior, and how modular arithmetic has given us a rigorous tool for simplifying the NFA into a DFA.

Resources and Usages

- Sipser, M. (2012). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.
  (Source of Figure 1.34 and the original NFA example)

- Graphviz. (2025). *Graph Visualization Software*. Retrieved from https://graphviz.org
  (Used to generate automaton diagrams)

- Python Software Foundation. (2025). *Python 3 Documentation*. Retrieved from https://www.python.org/doc/
  (Used for simulation code of the redesigned NFA)

- Microsoft. (2025). *Visual Studio Code*. Retrieved from https://code.visualstudio.com

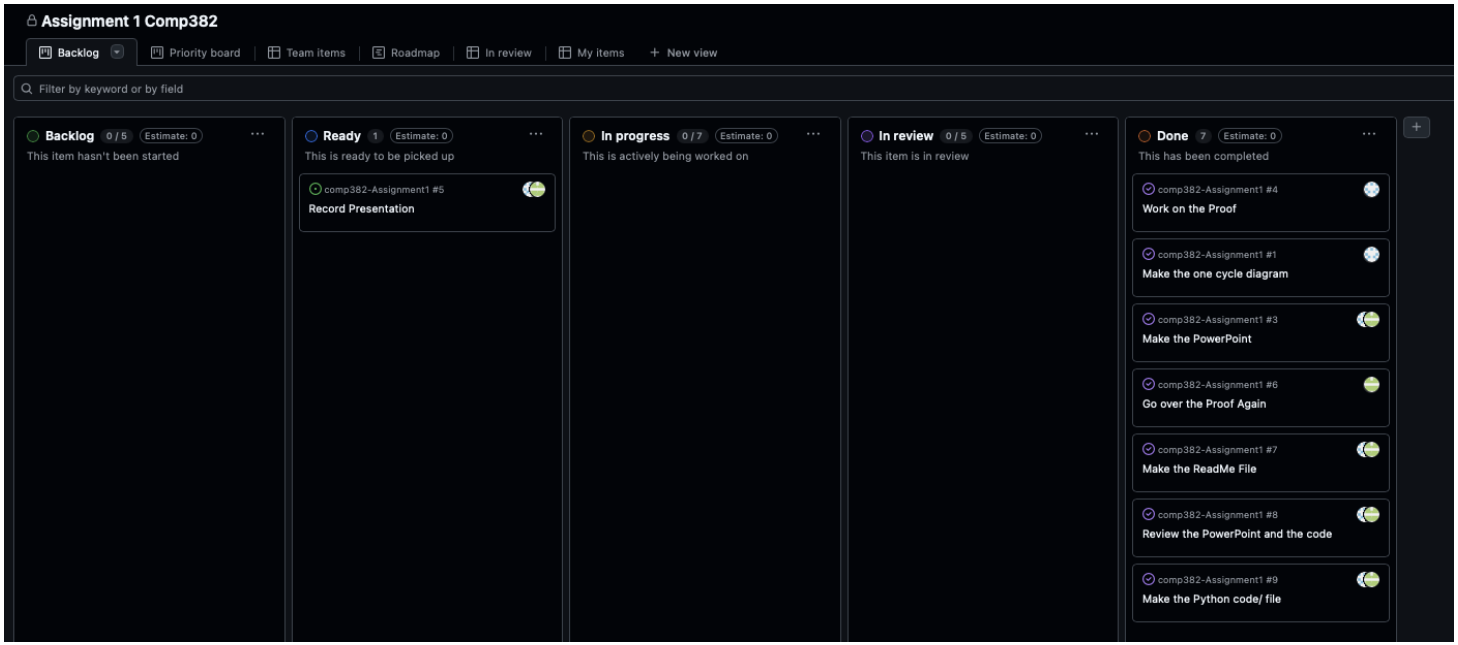
- GeeksforGeeks. (2024). *Modular Arithmetic and Applications*. Retrieved from https://www.geeksforgeeks.org/modular-arithmetic/
  (Background reference on modular arithmetic applied in automata design)

- Homebrew. (2025). *The Missing Package Manager for macOS (or Linux)*. Retrieved from https://brew.sh
  (Used as a package manager to install Graphviz via MacOS terminal)

- W3Schools. (2025). *Python % Operator*. Retrieved from https://www.w3schools.com/python/ref_mod.asp
  (Modulo operator explanation)

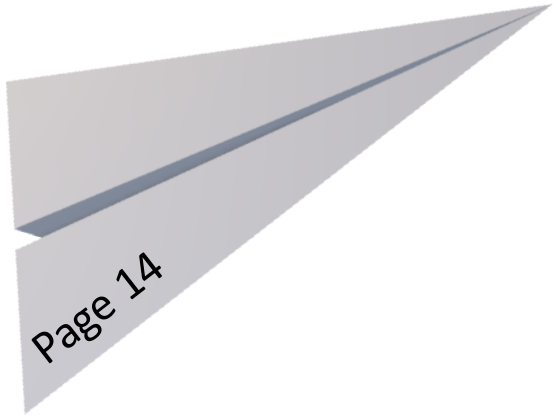
Logging and Misc.

- Github was used as a resource to assign tasks, which were then taken into discord for intermediary steps. For this, GitHub's project board feature was utilized, as seen below. Issues, and all other project information, are available for view through the public repository link found in "GitHub Link.txt"



Logging and Misc.

- Almost all communication about the project was done via discord. Main problem solving was solved synchronously via Discord voice calls, and asynchronously for smaller updates.
- GitHub also contains all deliverables for the project, for reproducibility purposes - PowerPoint, python, readme, and dot files.
- For Figure's 1 and 2, we utilized homebrew as a package manager to install Graphviz. Then, we created .dot files and ran them through Graphviz to produce the graphs.
- All portions of this presentations were tasked and split as follows, following no particular order:
  - NFA/DFA comparative python file – Sahil and Karan
  - Rough proof – Sahil
  - Finalized/ polished proof – Karan
  - Make the PowerPoint (Information and animations) - Sahil and Karan
  - Set up the GitHub/ create tasks - Karan
  - Make adjustments to GitHub – Sahil
  - Create README.txt file – Sahil and Karan
  - Upload deliverables to GitHub – Sahil
  - Find supplementary resources – Sahil and Karan
  - Create .dot files and Graphviz diagrams – Sahil
  - Testing redesigned NFA against empirical proof – Sahil and Karan



- Through this project, we gained practice in modular arithmetic, automata simplification, and using external tools (Discord, GitHub, Graphiz, Homebrew, and references) to create a reproducible workflow.