CPSC-406 Report

Erik Van Cruyningen
Chapman University

May 26, 2025

Abstract

Contents

1 Introduction **2**

2 Week by Week **2**
2.1 Week 1 . . . 2
2.1.1 Homework 1: Introduction to Automata . . . 2
2.1.2 Homework 1: DFAs and NFAs . . . 2
2.1.3 Exploration . . . 3
2.1.4 Questions and Comments . . . 3
2.2 Week 2 . . . 3
2.2.1 Homework 2: Automata . . . 3
2.2.2 Exploration . . . 5
2.2.3 Questions and Comments . . . 5
2.3 Week 3 . . . 5
2.3.1 Homework 3: Operations On Automata . . . 5
2.4 Week 4 . . . 6
2.4.1 Homework 4: Determinization . . . 6
2.4.2 Exploration . . . 7
2.4.3 Questions and Comments . . . 7
2.5 Week 5 . . . 7
2.5.1 Homework 5: ITALC Problems . . . 7
2.5.2 Exploration . . . 8
2.5.3 Questions and Comments . . . 8
2.6 Week 6 and 7 . . . 8
2.6.1 Homework 6: . . . 8
2.6.2 Exploration . . . 13
2.6.3 Questions and Comments . . . 14
2.6.4 Homework 7: . . . 14
2.6.5 Exploration . . . 14
2.6.6 Questions and Comments . . . 14
2.7 Week 8 and 9 . . . 14
2.8 Homework 8: Big O Notation Properties . . . 14
2.8.1 Exercise 1 . . . 14
2.8.2 Exercise 2 . . . 15
2.8.3 Exercise 5 . . . 16

2.9 Discussion . . . 16
2.10 Questions and Reflections . . . 16
2.11 Week 8 and 9 . . . 16
2.12 Homework 10 and 11 . . . 16
2.13 Homework 12 and 13 . . . 16

3 From Theory to Practice - Algorithm Analysis **18**

4 Evidence of Participation **19**

5 Conclusion **19**

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 Homework 1: Introduction to Automata

Problem 1: Characterize All Words to Sum 25 Cents

All words that end in the accepting state follow a pattern. They can be one of three things: Five fives, three fives and one ten, or one five and two tens. This can be written as:

$$(5^5)|(5^3 10)|(5^1 10^2)$$

This is every possible combination of valid answers, but answers over 25 cents are rejected.

Problem 2: Create a Regular Expression to express the pattern in the image

The pattern consists of any number of pushes and pays followed by at least one pay. Following the guidelines in the homework, the regex is:

$$(push + pay)^* pay$$

2.1.2 Homework 1: DFAs and NFAs

Problem 1: Determine whether the following words belong to L1, L2, or L3.

$$L1 := \{x01y \mid x, y \in \Sigma^*\}$$
$$L2 := \{w \mid |w| = 2^n, n \in \mathbb{N}\}$$
$$L3 := \{w \mid |w_0| = |w_1|\}$$

L1 accepts any word that contains "01" in it. L2 accepts words where the length is a power of 2. L3 accepts any word where the word is of equivalent length to the word w0.

	L1	L2	L3
w1 = 10011	1	0	1
w2 = 100	0	0	0
w3 = 10100100	1	1	0
w4 = 1010011100	1	0	0
w5 = 11110000	0	1	0

Problem 2: Determining Accepted States

We have a DFA defined as follows:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. Transition function:

$$\begin{aligned} &\delta(q_0, 0) = q_2, && \delta(q_0, 1) = q_0, \\ &\delta(q_2, 0) = q_2, && \delta(q_1, 1) = q_0, \\ &\delta(q_1, 0) = \delta(q_1, 1) = q_1 \end{aligned}$$

4. Initial state: q_0
5. Accepting state: $F = \{q_1\}$

For the words $w_1 = 0010$, $w_2 = 1101$, and $w_3 = 1100$, only w_1 and w_2 end in the accepted state q_1.

2.1.3 Exploration

I believe this material is included in the course because sets up the formal definitions of DFAs and NFAs that we can build off of later in the course. This material is quite interesting to me, because it seems you can build an automata for many things we use in our everyday life. My coffee maker has a selection of inputs, a beginning state, and a goal state, and I think it follows the definition of a DFA. Defining everyday objects as automata makes me understand coursework and comprehend the course better, as I can apply what we learn to everyday objects.

2.1.4 Questions and Comments

How common are objects in our lives that follow a DFA scheme? For example, an app on a phone could be seen as automata, and perhaps Instagram's "Goal State" would be keeping the app open for as long as possible.

2.2 Week 2

2.2.1 Homework 2: Automata

Problem 1: Word Processing with DFAs

Below is a table of example words and whether they are accepted by A1 and A2 as shown inside the homework.

w	A1	A2
w1 = AAA	0	1
w2 = AAB	1	0
w3 = ABA	0	0
w4 = ABB	0	0
w5 = BAA	0	1
w6 = BAB	0	0
w7 = BBA	0	0
w8 = BBB	0	0

Describe the language L(A(k)) accepted by A(k) for both k=1 and k=2:
A1 needs to start with an a, and then have an odd number of b's and end in a b.
A2 needs to end in at least two consecutive a's.

Problem 2: Implementing DFA runs

We begin by implementing the run method, which should begin by setting the current state to the initial state of the DFA. It then should go through all the characters in the passed in word, and update its state depending on the input and transition function. When it has finished running, it checks whether the current state is inside the list of final states and returns a boolean if it is in the list of final states.

```
def run(self, w):
      state = self.q0
      for char in w:
          state = self.delta[(state, char)]
      if state in self.F:
          return True
      return False
```

We then move on to defining the DFAs, which can be easily defined as a set of states, a list of accepted letters, a transition function outlining what to do with each input in each state, an initial state, and finally a set of accepting states. An example is:

```
A1 = dfa.DFA(
      Q={1, 2, 3},
      Sigma={"a", "b"},
      delta={
          (1, "a"): 2,
          (1, "b"): 4,
          (2, "a"): 2,
          (2, "b"): 3,
          (3, "a"): 2,
          (3, "b"): 2,
          (4, "a"): 4,
          (4, "b"): 4,
      },
      q0=1,
      F={3},
  )
```

Problem 3: (Designing DFAs

We implemented the DFAs into our Python program and examined what happened when we ran them with the list of all possible three letter combinations of a's and b's. All of them appeared to be correct and when we were implementing them, we saw that a lot of them shared a similar core structure and not much needed to be changed. For example, the DFA for "an odd number of a's and an odd number of b's" was identical to the DFA for "an even number of a's and an odd number of b's", except that the goal state was moved by one unit. This would apply to any even or odd counter. Another similarity was the words that contain a string of three characters, as they are identical except for the inputs.

Problem 4: (A new automata from an old one)

To make a DFA that accepts only the words that a different DFA rejects and rejects the one that the same DFA accepts, you simply need to take the original DFA and swap the accepting states and non accepting states. For example, if A0 only accepted state 3, then A1 could accept state 1 and 2 and it would be the exact opposite. The code for this follows this premise and finds all the states that are not in the original list of accepting states.

```
def refuse(self):
      N = set()
```

```
        for state in self.Q:
            if state not in self.F:
                N.add(state)

        # Create a new DFA that uses N
        newDFA = DFA(Q=self.Q, Sigma=self.Sigma, delta=self.delta, q0=self.q0, F=N)

        return newDFA
```

2.2.2 Exploration

I think this material is included in the course because it requires the students to critically think about the formal definitions of DFAs that we can use with either pure math or graphical depictions. I think it is quite cool how DFAs can be implemented so simplistically in code, and how practical they can be.

2.2.3 Questions and Comments

Is there a purpose to have the inverse of all DFAs? I understand many DFAs and their inverses have applications, but some DFAs must have an application and their inverse is completely useless

2.3 Week 3

2.3.1 Homework 3: Operations On Automata

Problem 1: Extended Transition Functions

The langauge accepted by A2 are words that are an odd number of characters in length and do not have any sequential b's.

If we look at our function, S(state, letter), we can recursively find what the state is for any given word.

$$S(1, abaa) = S(S(1, a), baa)$$

$$S(1, a) = 2$$

We then substitute to simplify:

$$S(2, baa) = S(S(2, b), aa)$$

$$S(2, b) = 4$$

Substitute again:

$$S(4, aa) = S(S(4, a), a)$$

$$S(4, a) = 2$$

Final State:

$$S(2, a) = 3$$

We then repeat the process for S2.

$$S(1, abba) = S(S(1, a), bba)$$

$$S(1, a) = 2$$

We then substitute to simplify:

$$S(2, bba) = S(S(2, b), ba)$$

$$S(2, b) = 1$$

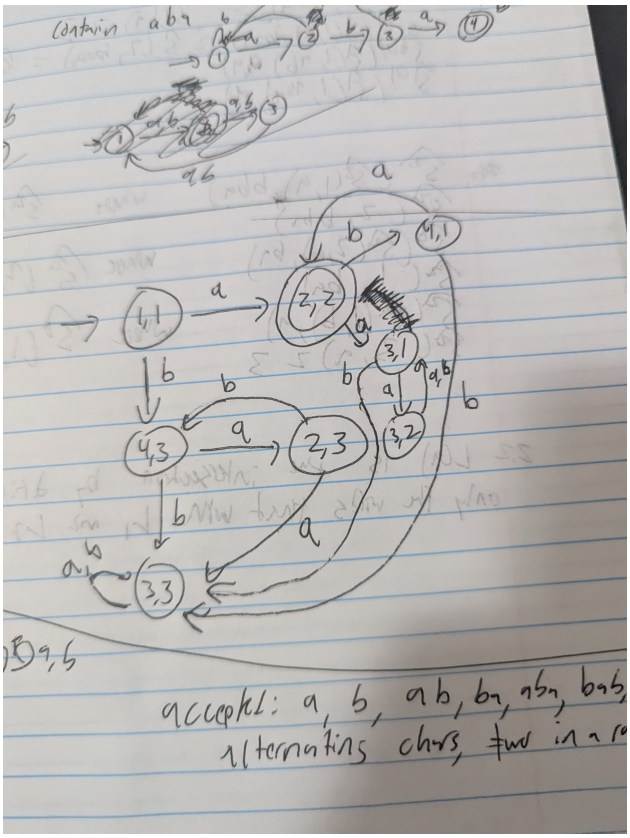

Figure 1: The $L(\mathcal{A}) = L(\mathcal{A}^{(1)}) \cap L(\mathcal{A}^{(1)})$

Substitute again:

$$S(1, ba) = S(S(1, b), a)$$

$$S(1, b) = 3$$

Final State:

$$S(3, a) = 3$$

We are sure that our new DFA, L3 is the combination of L1 and L2 because if captures every possible state of the two that is reachable. Every point in L3 is a set of the two states in L1 and L2, and it captures all possible routes that can make it to that place.

In order to change L3 such that is becomes the union of L1 and L2, you would need to make a new DFA that accepts all the words that both L1 and L2 accept.

2.4 Week 4

2.4.1 Homework 4: Determinization

Problem 1: Explain how a DFA can be seen as an NFA

A DFA can be seen as an NFA because the transition function can be changed from a mapping of a state and an input to be an input and a set of states that it leads to. A DFA will have singleton states

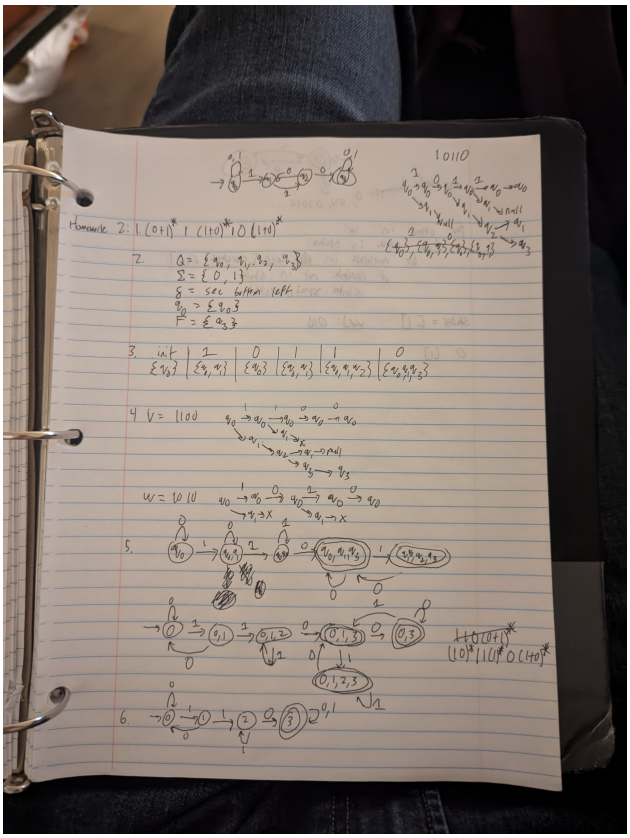

Figure 2: Homework 4, Problem 2

and the remaining features of the DFA stay the same in the new NFA. Because every DFA is an NFA by definition, L(A) will equal L(A') every time.

Problem 2: Describing NFAs. See Fig 2.

2.4.2 Exploration

I like this material in the course, because I can see how an NFA would be more useful than a DFA, but harder to implement in code. I am working on the first project, and it is quite stimulating to think about the changes required to alter an NFA to become a DFA.

2.4.3 Questions and Comments

Are there some things that regular expressions cannot solve? It is really good at pattern matching and parsing text, but I cannot imagine that it would be good at natural language text.

2.5 Week 5

2.5.1 Homework 5: ITALC Problems

All the problems are in image form, because the proofs would take a long time to type.

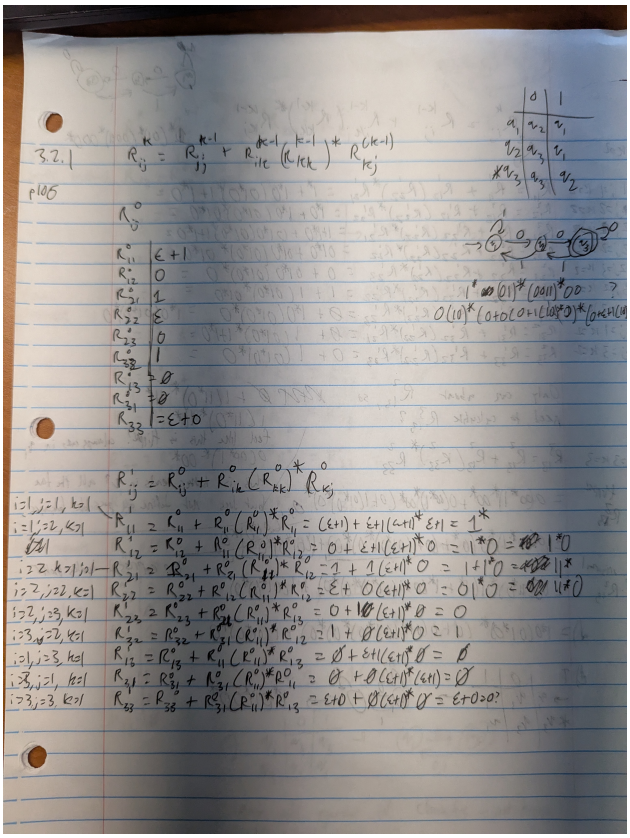

Figure 3: ITALC Problem 3.2.1 Part 1

2.5.2 Exploration

This material is quite interesting, but the proofs for RegEx are quite complicated. I am not too sure about the RegEx simplification, and I think I got some of the simplified formulas wrong. However it was very eye opening to work through these problems

2.5.3 Questions and Comments

Is there an easier method to prove that a DFA follows a regex structure? It takes a lot of work to probe a simple three state DFA, I cannot imagine it being feasible hand writing these formulas for more than four states.

2.6 Week 6 and 7

2.6.1 Homework 6:

Exercise A:

1. For an input 10^n, this Turing Machine returns 10^{n+1}

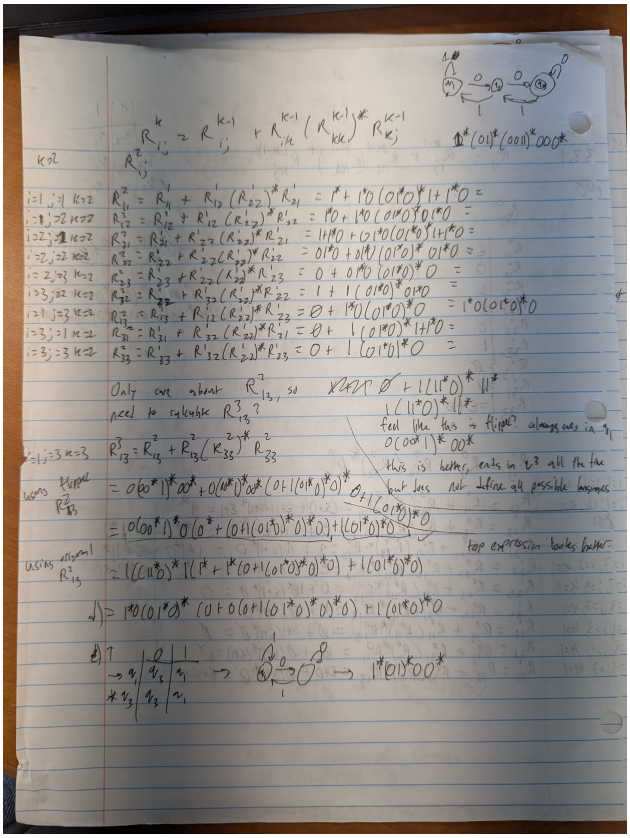

Figure 4: ITALC Problem 3.2.1 Part 2

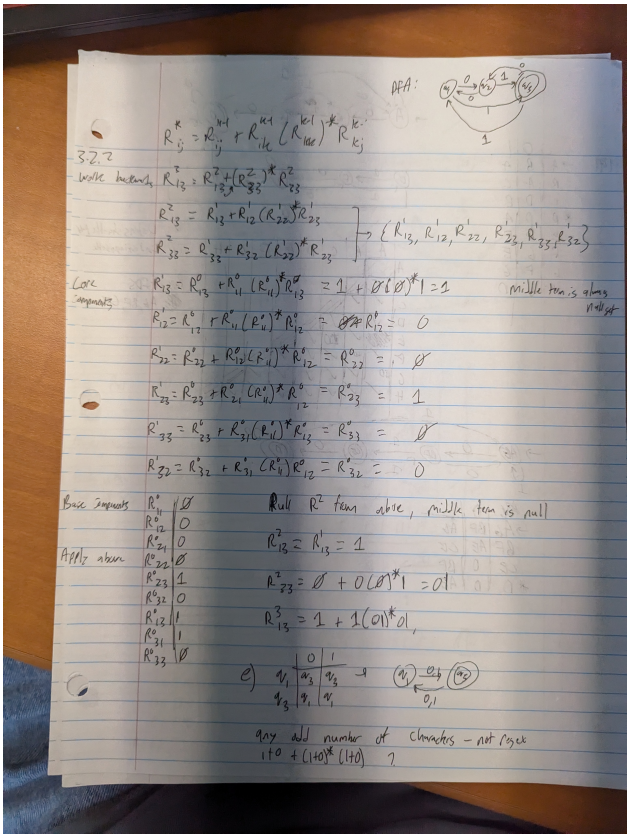

Figure 5: ITALC Problem 3.2.2

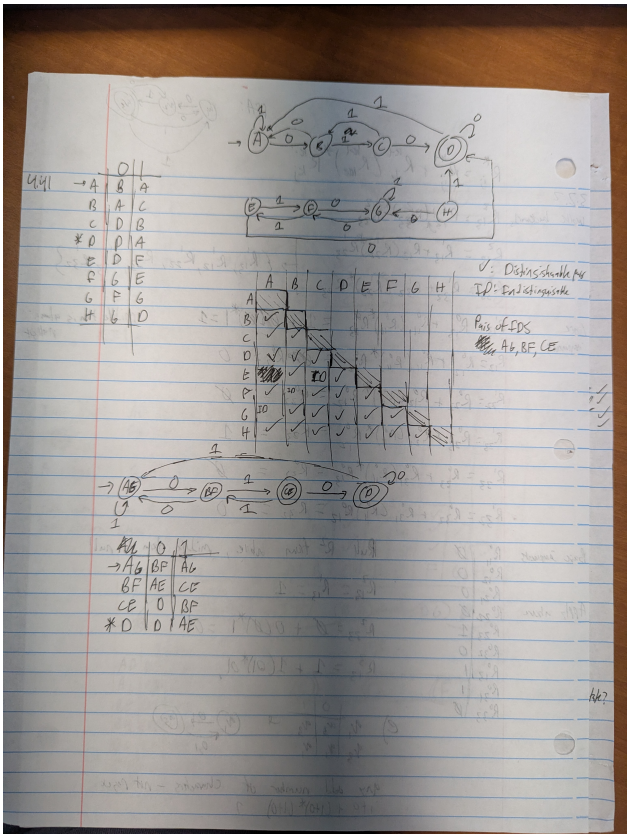

Figure 6: ITALC Problem 4.4.1

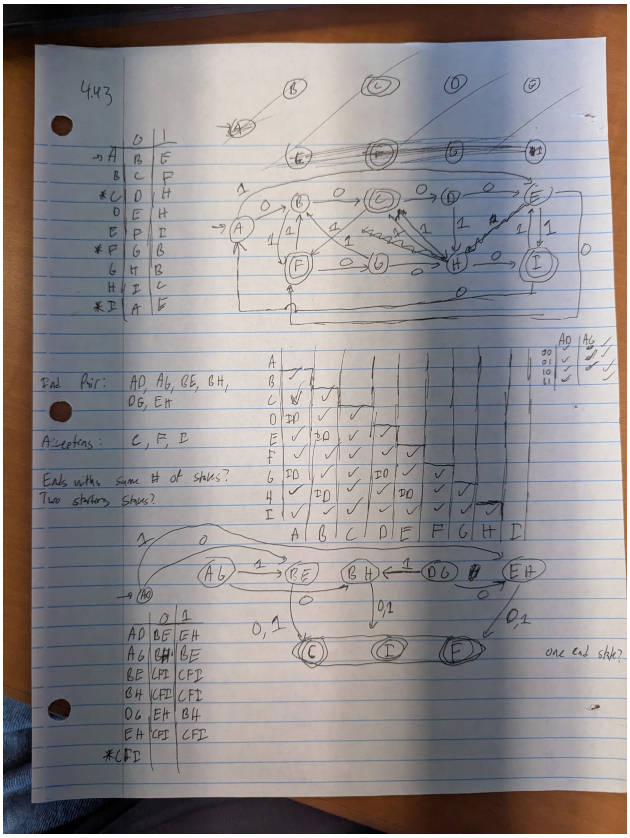

Figure 7: ITALC Problem 4.4.2

state	0	1	B
q0	X	q1,1,R	X
q1	q1,0,R	X	q2, 0, R
q2	X	X	X

2. For an input 10^n, this Turing Machine returns 1

state	0	1	B
q0	X	q1,1,R	X
q1	q1,B,R	X	q2, B, R
q2	X	X	X

3. For an input of a binary string, this returns the 1's swapped with 0's and the 0's swapped with 1's.

state	0	1	B
q0	q0,1,R	q1,0,R	q1,0,R
q1	X	X	X

Exercise B: Done for Extra Credit

1. If we want to take input 10^n10^m and output $10^n10^m10^{n+m}$, we would begin by writing the entire string to the output, write an extra 1, then go back to the beginning. Then, when you read a 0, go to the end, write a 0, then go back to the beginning. When you reach a 1, go to a new 0 (using new states), and then go to the end and write a 0. Repeat this process until you hit the next 1, and then copy this pattern for the second part. You will end up with $10^n10^m10^{n+m}$.

2. In order to double any binary string, like 111000, we could have one state for each digit in the string. We could read the first digit, go to the end of the string, write the digit, go back to the first digit, then go one digit right, and repeat. This would double 111000 to 111000111000, and would double any binary string.

Exercise C: Done for Extra Credit

These are the transition diagrams for Exercise A of this assignment. MA1 is exercise 1, MA2 is exercise 2, and MA3 is exercise 3.

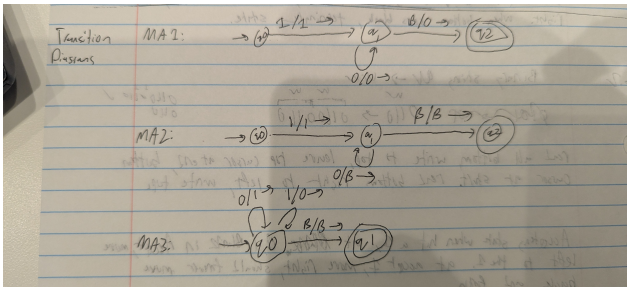

Figure 8: Transition Diagrams

2.6.2 Exploration

I wonder if we can make a hardware version of a Turing machine, and could we make it more powerful than a modern processor? We may have to ditch a physical tape method, but maybe a 1s and 0s bit stream could do something similar.

2.6.3 Questions and Comments

If we had a infinitely fast turing machine, what problems could we solve with it? For example, could we mine all remaining cryptocurrencies or solve the stock market? And can we create multiple constructions of turing machines that solve the same problem?

2.6.4 Homework 7:

Exercise 1:

Which of the following languages are decidable, recursively enumerable (r.e.), or have recursively enumerable complement (co-r.e.)?

1.
$$L_1 := \{M \mid M \text{ halts on itself}\}$$

2.
$$L_2 := \{(M, w) \mid M \text{ halts on the word } w\}$$

3.
$$L_3 := \{(M, w, k) \mid M \text{ halts on } w \text{ in at most } k \text{ steps}\}$$

1. L1 is not decidable because it is essentially the diagonal langauge. This means that our language would not be able to decide in finite time whether the input is accepting or rejecting. L1 is recursively enumerable because we can take a machine that takes in Mi, and then simulates M on Mi and accepts if M accepts. If this accepts, then our machine will accept Mi. This language is not co-r.e. because the complement would include both machines that do not accept Mi and machines that do not halt on Mi, and it would be impossible for us to determine which of these machines it is. This makes it not co-r.e.

2. L2 is not decidable because there can be no Turing machine that can decide for every M and w whether M(w) halts. This was proven by Turing himself. L2 is recursively enumerable because we can simulate every step of M(w). If M(w) halts, we will see it happen, and accept, and if it doesn't halt it will run forever. L2 is not co-r.e. because if our language does not halt on the word w, we will have no way of determining if the word is in the language. This means it cannot be co-r.e.

3. L3 is decidable because there is a specific bound to check if M halted. If M has not halted within that time, we can tell that the word is not in the language. It is also recursively enumerable and complement recursively enumerable because it is decidable.

2.6.5 Exploration

This makes me wonder if there are a finite set of decidable and undecidable problems. I would not know how to approach this problem, but it would be interesting to see if there are infinite problems.

2.6.6 Questions and Comments

Is the set of decidable problems infinite? Or what about the set of undecidable problems? If they are infinite, do we have simplification methods to reduce the size of things we have to compute?

2.7 Week 8 and 9

2.8 Homework 8: Big O Notation Properties

2.8.1 Exercise 1

- **Exercise 1: Order Functions by Growth Rate**
  Order the functions: 2^n, $e^{\log n}$, $\log(n)$, e^n, $e^{2\log(n)}$, $\log(\log(n))$, 2^{2n}, $n!$.

Simplify: $e^{\log n} = n$, $e^{2\log(n)} = n^2$
Slowest to Fastest:
- $\log(\log(n))$
- $\log(n)$
- n
- n^2
- 2^n
- e^n
- $n!$
- 2^{2^n}

2.8.2 Exercise 2

- **Property 1:** $f \in O(f)$

  Proof. By definition, $f \in O(f)$ if $\exists M, N$ such that $\forall n \geq N$, $f(n) \leq M \cdot f(n)$.
  Set $M = 1$, $N = 1$.
  Then $\forall n \geq 1$, $f(n) \leq 1 \cdot f(n)$, which is true since $f(n) = f(n)$.
  Thus, $f \in O(f)$. □

- **Property 2: If $c > 0$, then $O(c \cdot f) = O(f)$**

  Proof. $\underline{O(c \cdot f) \subseteq O(f)}$:
  Let $h \in O(c \cdot f)$.
  $\exists M_1, N_1$ such that $\forall n \geq N_1$, $h(n) \leq M_1 \cdot (c \cdot f(n)) = (M_1 \cdot c) \cdot f(n)$.
  Set $M = M_1 \cdot c$.
  Then $h(n) \leq M \cdot f(n)$, so $h \in O(f)$.
  $\underline{O(f) \subseteq O(c \cdot f)}$:
  Let $h \in O(f)$.
  $\exists M_2, N_2$ such that $\forall n \geq N_2$, $h(n) \leq M_2 \cdot f(n) = \left(\frac{M_2}{c}\right) \cdot (c \cdot f(n))$.
  Set $M = \frac{M_2}{c}$.
  Then $h(n) \leq M \cdot (c \cdot f(n))$, so $h \in O(c \cdot f)$.
  Thus, $O(c \cdot f) = O(f)$. □

- **Property 3: If $f(n) \leq g(n)$ for large n, then $O(f) \subseteq O(g)$**

  Proof. Let $h \in O(f)$.
  $\exists M_1, N_1$ such that $\forall n \geq N_1$, $h(n) \leq M_1 \cdot f(n)$.
  Given $f(n) \leq g(n)$ $\forall n \geq N_2$, set $N = \max(N_1, N_2)$.
  Then $\forall n \geq N$, $h(n) \leq M_1 \cdot f(n) \leq M_1 \cdot g(n)$.
  Thus, $h \in O(g)$, so $O(f) \subseteq O(g)$. □

- **Property 4: If $O(f) \subseteq O(g)$, then $O(f + h) \subseteq O(g + h)$**

  Proof. Let $k \in O(f + h)$.
  $\exists M_1, N_1$ such that $\forall n \geq N_1$, $k(n) \leq M_1 \cdot (f(n) + h(n))$.
  Since $O(f) \subseteq O(g)$, $\exists M_2, N_2$ such that $\forall n \geq N_2$, $f(n) \leq M_2 \cdot g(n)$.
  For $n \geq N_2$, $f(n) + h(n) \leq M_2 \cdot g(n) + h(n) \leq (M_2 + 1) \cdot (g(n) + h(n))$.
  Set $N = \max(N_1, N_2)$, $M = M_1 \cdot (M_2 + 1)$.

Then $k(n) \leq M \cdot (g(n) + h(n))$, so $k \in O(g + h)$.
Thus, $O(f + h) \subseteq O(g + h)$. □

- **Property 5: If $h(n) > 0 \ \forall n$, and $O(f) \subseteq O(g)$, then $O(f \cdot h) \subseteq O(g \cdot h)$**

  Proof. Let $k \in O(f \cdot h)$.
  $\exists M_1, N_1$ such that $\forall n \geq N_1$, $k(n) \leq M_1 \cdot (f(n) \cdot h(n))$.
  Since $O(f) \subseteq O(g)$, $\exists M_2, N_2$ such that $\forall n \geq N_2$, $f(n) \leq M_2 \cdot g(n)$.
  For $n \geq N_2$, $f(n) \cdot h(n) \leq M_2 \cdot g(n) \cdot h(n)$.
  Set $N = \max(N_1, N_2)$, $M = M_1 \cdot M_2$.
  Then $k(n) \leq M \cdot (g(n) \cdot h(n))$, so $k \in O(g \cdot h)$.
  Thus, $O(f \cdot h) \subseteq O(g \cdot h)$. □

2.8.3 Exercise 5

- **Pair 1: Bubble Sort vs. Insertion Sort**
  Bubble sort: worst, average $O(n^2)$, best $O(n)$ (optimized).
  Insertion sort: worst, average $O(n^2)$, best $O(n)$.
  Let $f(n) = n^2$ (bubble sort), $g(n) = n^2$ (insertion sort). By Property 1 ($f \in O(f)$), $O(f) = O(g) = O(n^2)$. Insertion sort often outperforms bubble sort in practice (fewer swaps).
- **Pair 2: Insertion Sort vs. Merge Sort**
  Insertion sort: worst, average $O(n^2)$, best $O(n)$.
  Merge sort: all cases $O(n \log n)$.
  Let $f(n) = n^2$ (insertion sort), $g(n) = n \log n$ (merge sort). Since $n \log n \leq n^2$, by Property 3, $O(n \log n) \subseteq O(n^2)$. Merge sort is asymptotically faster.
- **Pair 3: Merge Sort vs. Quick Sort**
  Merge sort: all cases $O(n \log n)$.
  Quick sort: best, average $O(n \log n)$, worst $O(n^2)$.
  Let $f(n) = n \log n$ (merge sort), $g(n) = n \log n$ (quick sort average). By Property 1, $O(f) = O(g) = O(n \log n)$. Quick sort's worst case is $O(n^2)$, but it's often faster in practice (better cache locality).

2.9 Discussion

These proofs solidify the understanding of Big O notation as a tool for analyzing algorithm complexity, connecting asymptotic analysis to practical performance evaluation.

2.10 Questions and Reflections

How do these properties extend to other asymptotic notations like Θ or Ω? Are there cases where Big O comparisons fail to capture practical differences in algorithm performance?

2.11 Week 8 and 9

2.12 Homework 10 and 11

2.13 Homework 12 and 13

Homework 12: I solved this problem using diagrams on paper, but I do not know how to write it up into LaTeX. I got the optimal solution to be a flow rate of 19 with a minimum cut of the graph to be 19 as well. You can obtain 19 if you cut s->a and s->b or c->t and d->t. This follows the convention of maximal flow, minimal cut, which shows that our minimal cut will be equal to our maximum flow. These cuts are not necessarily unique, as I have shown two that are different above.

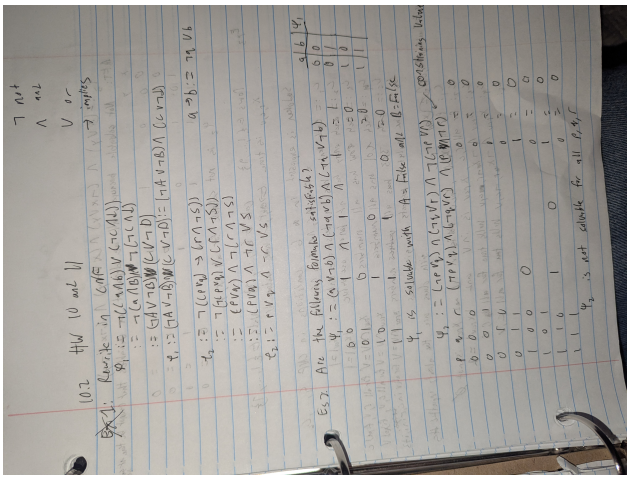

Figure 9: HW 10

Homework 13:

The algorithm counts executions of line 5 across three nested loops:

$$\sum_{k=1}^{n-1} \sum_{l=k+1}^{n} l = \sum_{k=1}^{n-1} \left[\sum_{l=1}^{n} l - \sum_{l=1}^{k} l \right] \tag{1}$$

$$= \sum_{k=1}^{n-1} \left[\frac{n(n+1)}{2} - \frac{k(k+1)}{2} \right] \tag{2}$$

$$= (n-1) \cdot \frac{n(n+1)}{2} - \frac{1}{2} \sum_{k=1}^{n-1} k(k+1) \tag{3}$$

Expanding $k(k+1) = k^2 + k$ and using the given formulas:

$$= (n-1) \cdot \frac{n(n+1)}{2} - \frac{1}{2} \left[\frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} \right] \tag{4}$$

$$= (n-1)n \left[\frac{n+1}{2} - \frac{2n-1}{12} - \frac{1}{4} \right] \tag{5}$$

$$= (n-1)n \left[\frac{6(n+1) - (2n-1) - 3}{12} \right] \tag{6}$$

$$= (n-1)n \left[\frac{4n+4}{12} \right] = \frac{n(n-1)(n+1)}{3} \tag{7}$$

The algorithm returns $\frac{n(n-1)(n+1)}{3}$.

2. Time Complexity

The algorithm has three nested loops with bounds:

- Outer loop: k from 1 to $n-1$ (runs $n-1$ times)
- Middle loop: l from $k+1$ to n (runs at most n times per outer iteration)

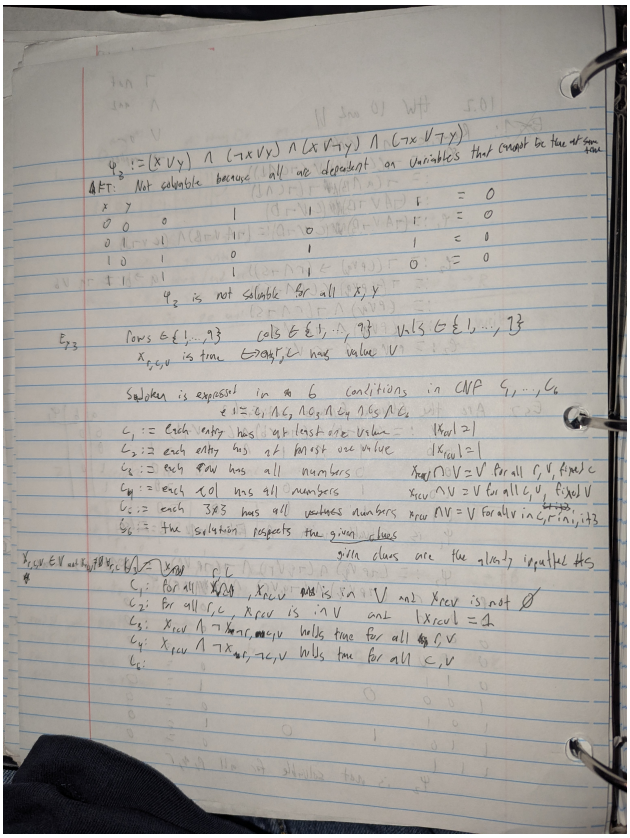

Figure 10: HW 11

- Inner loop: m from 1 to l (runs at most n times per middle iteration)

Total operations: $(n-1) \times n \times n = O(n^3)$. The worst-case running time is $O(n^3)$.

3 From Theory to Practice - Algorithm Analysis

Looking back on this course, I realize it took me on quite a journey through computation theory and problem-solving techniques. What started with pretty abstract concepts ended up connecting to things I could actually build and use.

At the beginning, we dove DFAs and NFAs. I'll be honest—at first, these felt pretty theoretical and disconnected from "real" programming. But as we worked through examples, I started to see how these machines could recognize patterns and handle regular languages. The surprising thing I learned was that even though DFAs and NFAs look structurally different, they're actually equivalent in terms of what they can compute. Both can recognize exactly the same set of regular languages, which was one of those realization moments for me.

Then we moved on to Turing Machines, and suddenly the scope got much bigger. These machines can handle any computation that's algorithmically possible, which is pretty eye opening. This is where we started talking about decidable problems—ones that can actually be solved by an algorithm—versus undecidable problems that simply can't be solved, no matter how clever we get. It was fascinating to learn that there are fundamental limits to what computation can achieve.

The most rewarding part of the course was definitely our coding project,, where we built a Sudoku solver using an SAT solver. This finally brought everything together in a concrete way. We had to take the rules of Sudoku and encode them as a Boolean formula in CNF format, which essentially transformed the puzzle into a Boolean satisfiability problem. Using the SAT solver to find solutions felt odd at first, but I came to understand how it systematically searches for variable assignments that satisfy all the constraints.

By the end of the course, I felt like I'd traveled from abstract mathematical models all the way to hands-on problem solving. The progression made sense—starting with simple automata, building up to the full power of Turing machines, exploring the boundaries of what's computationally feasible, and finally applying those insights to solve actual problems. It wasn't just about understanding computational theory anymore; I'd actually experienced how that theory translates into practical algorithmic solutions.

4 Evidence of Participation

Over the course of this semester, I asked all my questions in Discord, and went to one seminar that I forgot to write about in the Discord. My questions were as follows:

- 1. How common are objects in our lives that follow a DFA scheme?
- 2. Do all DFAs that have a purpose have a meaningful inverse? Or is there only a small subset of meaningful DFAs that have a meaningful inverse?
- 3. Are there some things that regular expressions are poor at doing?
- 4. What are the limits of regular expressions?
- 5. If we had a infinitely fast turing machine, what problems could we solve with it? For example, could we mine all remaining cryptocurrencies or solve the stock market?
- 6. Can we create multiple constructions of turing machines that solve the same problem?
- 7. Is there an easier method to prove that a DFA follows a regex structure?
- 8. Can make a hardware version of a Turing machine, and could we make it more powerful than a modern processor?
- 9. Is the set of decidable problems infinite? Or what about the set of undecidable problems?

The seminar I went to was a speaker event about the use of AI by hiring managers in the job market. It was presented by a professor at Chapman University who works in the business school and was doing analytics. It was really interesting to see how important keywords are for applying to jobs. She also spoke about how people who graduate during a recession tend to have a lower lifetime household income. It was a very interesting event, and afterwards I felt inspired to apply to more jobs and secure work over the summer.

5 Conclusion

Looking back on this course, I can see that I've learned both foundational theory and its practical applications. When we started with deterministic and nondeterministic finite automata (DFAs and NFAs), I wasn't sure how these abstract machines would be relevant to my work as a software engineer or computer scientist. They seemed removed from the practical problems I usually encounter. As the course continued, though, I came to understand that these models provide a framework for thinking systematically about computation and problem solvability.

When we covered Turing machines, we examined which problems are computable and learned to distinguish between problems we can solve algorithmically and those that are inherently undecidable, such as the Halting Problem. This changed how I approach problems. Instead of immediately jumping into coding, I now consider

whether a problem is actually solvable before beginning implementation. This approach has proven useful when working with complex systems, validating inputs, or designing algorithms for unusual cases.

The section on NP-completeness was particularly valuable. We studied how to classify problems by computational difficulty and learned that certain problems, while easy to verify, may not be efficiently solvable. This concept became concrete during our Sudoku solver project using a SAT solver. We converted Sudoku rules into a Boolean satisfiability problem—an NP-complete problem—and used a SAT solver to find solutions. This project tied together the theoretical concepts we had studied in a practical way and helped solidify my understanding of the material.

Now that the course is complete, I can see how algorithm analysis applies to various areas of software engineering and computer science. When working on data validation, optimizing backend systems, or evaluating new features, I find myself considering questions like: Is this problem computable? Is it tractable? Should I use a heuristic or approximation? These questions stem from the theory we covered and have influenced how I approach development work.

Overall, this course has taught me not only how to solve problems, but also which problems can and cannot be solved which gives me a useful perspective to carry forward.