CPSC-406 Report

Ethan Tapia
Chapman University

May 22, 2025

Abstract

This report documents my journey through CPSC-406, focusing on the theoretical foundations of computation and algorithms. Beginning with the fundamentals of finite automata, the report progresses through deterministic and non-deterministic finite automata, regular expressions, context-free grammars, Turing machines, computability theory, and the complexity classes P and NP. Each section contains detailed solutions to weekly homework assignments, demonstrating my understanding and application of these concepts. The synthesis section explores the connections between complexity theory and practical algorithm design, while reflecting on the limitations of computational models. Throughout the course, I've developed not only technical skills in analyzing algorithms but also an appreciation for the elegant mathematical frameworks that underpin computer science.

Contents

1 Introduction

This report chronicles my semester-long exploration of the theoretical foundations of computer science in CPSC-406. The course has taken me from the basic building blocks of computation—finite automata and regular languages—to the frontiers of what computers can and cannot compute efficiently.

Beginning with deterministic finite automata (DFAs), I learned how to model simple computational processes and recognize regular languages. The progression to non-deterministic finite automata (NFAs) and their equivalence to DFAs demonstrated how different computational models can express the same language class. Regular expressions provided an elegant notation for describing patterns, while the subset construction and minimization algorithms revealed the practical aspects of implementing language recognizers.

As the course advanced, we explored more powerful computational models: context-free grammars, pushdown automata, and ultimately Turing machines. These models set the stage for understanding fundamental questions about computability—what problems can be solved algorithmically—and complexity—how efficiently such solutions can be implemented.

The analysis of algorithms formed a central theme, with asymptotic notation (Big O, Theta, Omega) providing a framework for comparing algorithm efficiency regardless of hardware specifics. The distinction between polynomial-time and exponential-time algorithms highlighted the practical importance of efficiency, while NP-completeness revealed a fascinating class of problems that resist efficient solutions despite their solutions being easily verifiable.

Network flow problems and Boolean satisfiability offered concrete examples of important computational problems with wide-ranging applications. The Ford-Fulkerson algorithm and SAT solvers demonstrated how theoretical concepts translate into practical algorithms for solving real-world problems.

Throughout this report, each weekly section documents not only the solutions to assigned problems but also my growing understanding of the theoretical landscape of computation. The synthesis section ties together these threads, reflecting on how this theoretical foundation informs my approach to algorithm design and analysis in practice.

2 Week by Week

2.1 Week 1

Lecture Summary

A finite automaton consists of a finite set of **states** (Q), an **alphabet** (Σ), a **transition function** (δ), a **starting state** (q_0), and a set of **accepting states** (F).

It can be formally represented as:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- Q is the set of states,
- Σ is the input alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states.

2.2 Week 2

Lecture Summary

We defined *deterministic finite automata* (DFAs) more formally and proved basic properties: every DFA has exactly one outgoing transition per symbol in Σ, and DFAs are closed under union, intersection, and complement. We practiced writing DFAs in code by constructing the transition table and simulation routine. We concluded with a discussion of the state minimization and Myhill-Nerode equivalence classes.

Homework 1

Problem 1: Characterizing Accepted Sequences

The given problem involves designing a finite automaton that accepts sequences of 5 and 10-cent inputs summing to 25 cents.

Solution: We define the equation:

$$5a + 10b = 25 \tag{1}$$

where a is the number of 5-cent inputs and b is the number of 10-cent inputs. Solving for valid pairs:

- $(a = 5, b = 0) \Rightarrow$ Sequence: $5, 5, 5, 5, 5$
- $(a = 3, b = 1) \Rightarrow$ Sequence: $5, 5, 5, 10$
- $(a = 1, b = 2) \Rightarrow$ Sequence: $5, 10, 10$

These sequences are precisely those accepted by the automaton. The machine accepts a sequence if the total sum equals 25 cents.

Problem 2: Defining Valid Variable Names

A valid variable name must begin with a letter (ℓ) and be followed by any number of letters or digits (d).

Regular Expression:

$$\ell(\ell|d)^* \tag{2}$$

Solution: *Finite Automaton:* - States: q_0 (initial), q_1 (accepting). - Transitions: - $q_0 \to q_1$ on input ℓ - $q_1 \to q_1$ on input ℓ or d

Problem 3: Classification of Words in L_1, L_2, L_3

The given languages are defined as follows:

- $L_1 = \{x0y \mid x, y \in \Sigma^*\}$: The set of words that contain at least one '0'.
- $L_2 = \{w \mid |w| = 2^n \text{ for some } n \in \mathbb{N}\}$: The set of words whose length is a power of 2.
- $L_3 = \{w \mid |w|_0 = |w|_1\}$: The set of words where the number of 0s equals the number of 1s.

Solution: We analyze each word based on these conditions:

	L_1	L_2	L_3
$w_1 = 10011$	✓		
$w_2 = 100$	✓		
$w_3 = 10100100$	✓	✓	
$w_4 = 1010011100$	✓		✓
$w_5 = 11110000$	✓	✓	✓

Table 1: Classification of words into L_1, L_2, L_3

Problem 4: DFA Analysis

Given the DFA with states q_0 (start), q_2, and q_1 (accepting), we determine which words end in the accepting state q_1.

Transitions:

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

Checking Words:

- $w_1 = 0010$: $q_0 \to q_2 \to q_2 \to q_1 \to q_1$ ✓(Accepted)
- $w_2 = 1101$: $q_0 \to q_0 \to q_0 \to q_2 \to q_1$ ✓(Accepted)
- $w_3 = 1100$: $q_0 \to q_0 \to q_0 \to q_2 \to q_2$ (Not Accepted)

Solution:

$w_1 = 0010 \to$ ✓*Accepted*
$\mathrm{w}_2 = 1101 \to$ ✓*Accepted*
$\mathrm{w}_3 = 1100 \to$ *Rejected*

This confirms that w_1 and w_2 end in the accepting state, while w_3 does not.

Chapter 2.1 Report:

Chapter 2.1 discusses the use of finite automata in modeling real-world protocols, particularly in the context of electronic money transactions. The section introduces a three-party system involving a customer, a store, and a bank. The goal is to ensure that digital money is not duplicated or reused fraudulently.

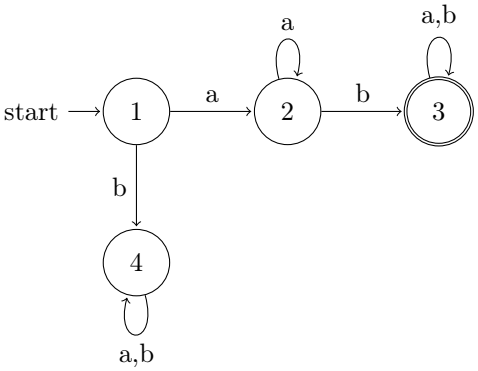
The protocol consists of five primary actions: pay, cancel, ship, redeem, and transfer. Each party's behavior is modeled using finite automata to track transaction states. The section highlights how such models can reveal vulnerabilities—such as a store shipping goods before verifying payment—showcasing the importance of automata in validating protocol security.

Finite automata prove to be useful for detecting logical flaws in transaction systems, ensuring valid sequences of operations. The chapter serves as an introduction to the application of formal computational models in the security and validation of protocols.

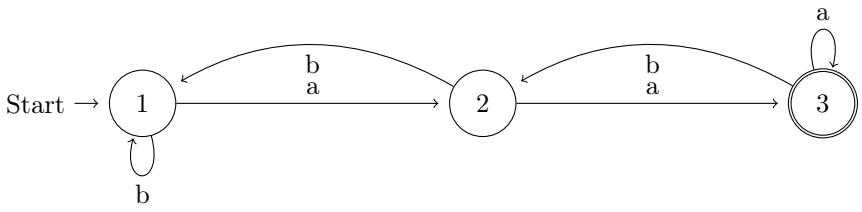
2.3 Week 3

Lecture Summary
We showed how to build new automata from old via *union*, *concatenation*, and *Kleene-star*. Given DFAs A and B, their union DFA simulates both in parallel using the product-state construction. Concatenation and star constructions use nondeterministic gadgets (or extra ε–transitions). We also contrasted NFAs vs. DFAs and previewed the subset-construction for determinization.

Homework 2 Exercise 2: Implementing DFA Runs



DFA A_1



DFA A_2

Words accepted or refused by $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively

w	Accepted A_1	Accepted A_2
aaa	×	✓
aab	✓	×
aba	×	×
abb	×	×
baa	×	✓
bab	×	×
bba	×	×
bbb	×	×

The table above summarizes the words accepted or rejected by DFA A_1 and DFA A_2. To implement these automata *programmatically*, we define the DFA class in `dfa.py`, which allows us to process input words according to their respective state transition diagrams.

DFA Implementation in `dfa.py`

This introduction describes the design of the `dfa.py`, consisting of:

- Q - a finite set of states.
- Σ - an input alphabet.
- $\delta : Q \times \Sigma \to Q$ - a transition function.
- $q_0 \in Q$ an initial state.
- $F \subseteq Q$ - a set of final accepting states.

The `DFA` class constructor takes these five elements (`Q`, `Sigma`, `delta`, `q0`, and `F`), using the method:

- `run(w)`: Runs the DFA on input string `w` and determines whether or not `w` is accepted based on the state it finishes at.

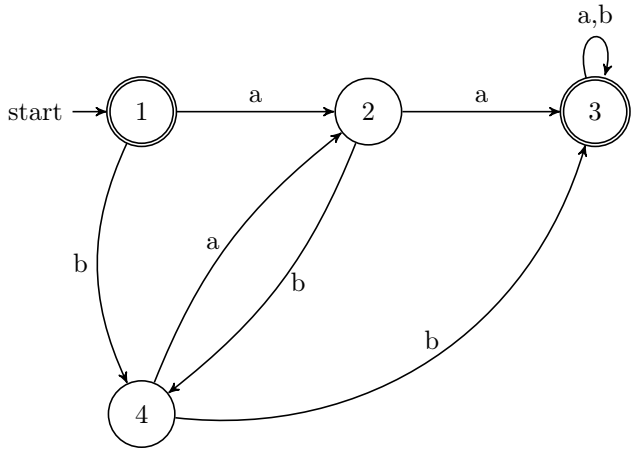
Implementation In the following code snippet, the `run` method processes the symbols of the input `w` sequentially, looking up the next state based on the current state and the input symbol. If an invalid transition is encountered, the method immediately returns `False`. Otherwise, if the DFA ends in a state

that is a member of `F`, `True` is returned (meaning `w` is accepted); if it ends in some other state, `False` is returned.

```
class DFA :

    # init the DFA
    def __init__(self, Q, Sigma, delta, q0, F) :
        self.Q = Q # set of states
        self.Sigma = Sigma # set of symbols
        self.delta = delta # transition function
        self.q0 = q0 # initial state
        self.F = F # final states

   # print the data of the DFA
    def __repr__(self) :
        return f"DFA({self.Q},\n\t{self.Sigma},\n\t{self.delta},\n\t{self.q0},\n\t{self.F})"

    # run the DFA on the word w
    # return if the word is accepted or not
    # modify as needed
    def run(self, w) :
        # todo
        # start at initial state
        current_state = self.q0
        for symbol in w:
            if (current_state, symbol) in self.delta:
                current_state = self.delta[(current_state, symbol)]
            else:
                # invalid transition (dead state)
                return False
            # accept if in final state
        return current_state in self.F
```

Exercise 4: A new automaton from an old one

Below is DFA A_0 which accepts exactly the words that A refuses and vice versa.



In A_0, nodes 1 and 3 are the accepting final states, while nodes 2 and 4 are normal states.

Exercise 2.2.4: DFAs over $\{0, 1\}$

(a) The set of all strings ending in 00

DFA Description:

$$
\begin{aligned}
Q &= \{ q_0, q_1, q_2 \}, \\
\Sigma &= \{0, 1\}, \\
\delta &\text{ is given by the table below,} \\
q_0 &\text{ is the start state,} \\
F &= \{ q_2 \}.
\end{aligned}
$$

Transition Table:

δ	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_0

Explanation:

- q_0 - we have not yet seen a trailing zero
- q_1 - the string currently ends in exactly one zero
- q_2 (accepting) - the string ends in at least two consecutive zeros

(b) The set of all strings with three consecutive 0s

DFA Description:

$$
\begin{aligned}
Q &= \{ q_0, q_1, q_2, q_3 \}, \\
\Sigma &= \{0, 1\}, \\
\delta &\text{ is given by the table below,} \\
q_0 &\text{ is the start state,} \\
F &= \{ q_3 \}.
\end{aligned}
$$

Transition Table:

δ	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_3	q_0
q_3	q_3	q_3

Explanation:

- q_0 - we have seen 0 consecutive zeros so far
- q_1 - we have seen exactly 1 consecutive zero
- q_2 - we have seen exactly 2 consecutive zeros
- q_3 (accepting) - we have seen at least 3 consecutive zeros

(c) The set of all strings with `011` as a substring

DFA Description:

$$
\begin{aligned}
&Q = \{\, q_0,\, q_1,\, q_2,\, q_3\},\\
&\Sigma = \{0, 1\},\\
&\delta \text{ is given by the table below,}\\
&q_0 \text{ is the start state,}\\
&F = \{\, q_3\}.
\end{aligned}
$$

Transition Table:

δ	0	1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_3	q_3

Explanation:

- q_0 - no partial match yet
- q_1 - we matched a single `0`
- q_2 - we matched `01`
- q_3 (accepting) - we found `011` somewhere in the string

2.4 Week 4

Lecture Summary
We introduced *nondeterministic finite automata* (NFAs), which allow multiple or zero δ–transitions on a given symbol (and ε–moves). The key theorem: for every NFA there is an equivalent DFA obtained by the subset construction. We worked through examples, proving that NFAs and DFAs recognize the same class of regular languages. Efficiency remarks: NFAs can be exponentially smaller, but DFAs run in linear time per input symbol.

Homework 3

Problem 1: Extended transition function
Consider two DFAs:

$$\mathcal{A}^{(1)} = \big(Q^{(1)}, \Sigma, \delta^{(1)}, q_0^{(1)}, F^{(1)}\big) \quad \text{and} \quad \mathcal{A}^{(2)} = \big(Q^{(2)}, \Sigma, \delta^{(2)}, q_0^{(2)}, F^{(2)}\big),$$

over the alphabet $\Sigma = \{a, b\}$.

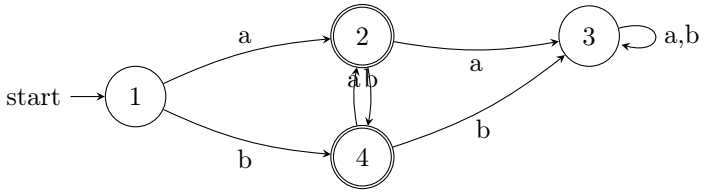


Diagram for $A^{(1)}$:

- State 1 is the initial state (arrow from the left).
- From state 1: reading 'a' goes to 2; reading 'b' goes to 4.

- From state 2: reading 'a' goes to 3; reading 'b' goes to 4.
- From state 4: reading 'a' goes to 2; reading 'b' goes to 3.
- From state 3: reading either 'a' or 'b' loops on 3.
- States 2 and 4 are accepting, while state 3 is a non-accepting sink.

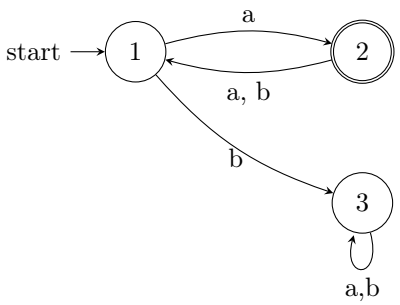


Diagram for $A^{(2)}$:

- State 1 is the initial state (arrow from the left).
- From state 1: reading 'a' goes to 2; reading 'b' goes to 3.
- From state 2: reading 'a' or 'b' goes back to 1.
- State 3 loops on both 'a' and 'b'.
- State 2 is the only accepting state.

a. Descriptions of Accepted Languages

- $\mathcal{A}^{(1)}$**:** Accepts non-empty words in which no two consecutive letters are the same.
- $\mathcal{A}^{(2)}$**:** Accepts words of odd length where every letter at an odd position of w is the letter a.

Hence,

$$L\big(A^{(2)}\big) = \big\{\, w \in \{a,b\}^* \mid |w| \text{ is odd and the odd-indexed letters are all } a \big\}.$$

b. Computation: $\widehat{\delta}^{(1)}\big(1, \texttt{abaa}\big)$

The automaton $A^{(1)}$ has states $\{1,2,3,4\}$, where:

- State 1 is the initial state (non-final).
- State 2 indicates the last letter read was a (no violation).
- State 4 indicates the last letter read was b (no violation).
- State 3 is the "sink" or "trap" state (once a violation occurs, e.g. two consecutive letters the same).

We compute step-by-step for the input `abaa`:

$$\widehat{\delta}^{(1)}(1, \texttt{abaa}) :$$

1. Start in state 1, input = `abaa`.
2. Read 'a': $\delta^{(1)}(1,a) = 2$. Remaining input = `baa`.
3. Now in state 2, read 'b': $\delta^{(1)}(2,b) = 4$. Remaining input = `aa`.

4. Now in state 4, read 'a': $\delta^{(1)}(4, a) = 2$. Remaining input = `a`.
5. Now in state 2, read 'a': $\delta^{(1)}(2, a) = 3$. Remaining input is empty.

Therefore,

$$\widehat{\delta}^{(1)}(1, \texttt{abaa}) \;=\; 3.$$

Since state 3 is the non-accepting sink, the string `abaa` is not accepted by $A^{(1)}$.

b. Computation: $\widehat{\delta}^{(2)}\big(1, \texttt{abba}\big)$

Automaton $A^{(2)}$ can be thought of as having states:

- State 1: an even number of letters read so far (initial, non-final).
- State 2: an odd number of letters read so far, with "odd positions must be a" still satisfied (accepting).
- State 3: dead/trap state (if the condition on odd positions is violated).

We compute for `abba` (positions are 1,2,3,4):

$$\widehat{\delta}^{(2)}(1, \texttt{abba}) :$$

1. Start in state 1, input = `abba`.
2. Read 'a' (position 1): $\delta^{(2)}(1, a) = 2$. Remaining input = `bba`.
3. Now in state 2, read 'b' (position 2): $\delta^{(2)}(2, b) = 1$. Remaining input = `ba`.
4. Now in state 1, read 'b' (position 3): $\delta^{(2)}(1, b) = 3$. Remaining input = `a`.
5. Now in state 3, read 'a': $\delta^{(2)}(3, a) = 3$. Remaining input is empty.

Hence,

$$\widehat{\delta}^{(2)}(1, \texttt{abba}) = 3,$$

and state 3 is non-accepting. Therefore, `abba` is not accepted by $A^{(2)}$.

Problem 2: Product automata

We now define a *product automaton* A from $A^{(1)}$ and $A^{(2)}$ in order to recognize

$$L\big(A^{(1)}\big) \;\cap\; L\big(A^{(2)}\big).$$

a. Construct the intersection automaton A

States. The state set of A is the Cartesian product

$$Q \;=\; Q^{(1)} \;\times\; Q^{(2)}.$$

If $Q^{(1)} = \{1, 2, 3, 4\}$ and $Q^{(2)} = \{1, 2, 3\}$, then

$$Q = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3), (4,1), (4,2), (4,3)\}.$$

Initial state. $(1, 1)$, since 1 is the initial state of $A^{(1)}$ and 1 is the initial state of $A^{(2)}$.

Transition function. For $(p, q) \in Q$ and $x \in \{a, b\}$,

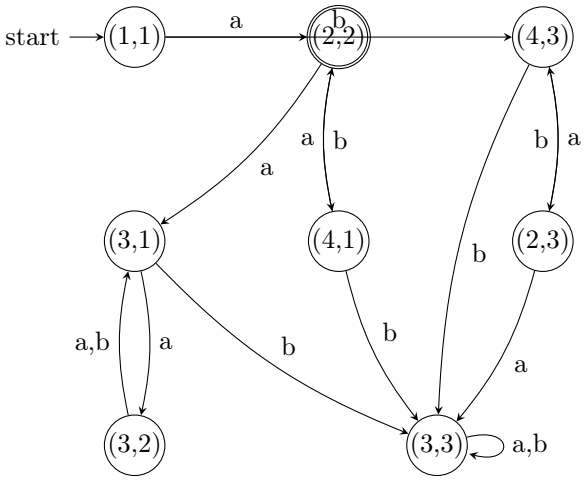
$$\delta\big((p, q), x\big) \;=\; \big(\delta^{(1)}(p, x),\; \delta^{(2)}(q, x)\big).$$

Final states (for intersection). A pair (p, q) is final if $p \in F^{(1)}$ and $q \in F^{(2)}$. - For $A^{(1)}$, we have $F^{(1)} = \{2, 4\}$ (all non-empty strings with no two consecutive letters the same). - For $A^{(2)}$, we have $F^{(2)} = \{2\}$ (odd-length strings with a in every odd position).

Thus

$$F \;=\; \{(p, q) \mid p \in \{2, 4\},\, q = 2\}.$$

Diagram of product automaton A showing all transitions. For brevity, not every state is drawn if some are unreachable.



b. Why $L(A) = L\big(A^{(1)}\big) \cap L\big(A^{(2)}\big)$

A string w is accepted by A precisely if:

- Following the transitions of A on w leads from $(1, 1)$ to a final state (p, q).
- This happens exactly when p is a final state of $A^{(1)}$ *and* q is a final state of $A^{(2)}$.

Thus w is accepted by both $A^{(1)}$ and $A^{(2)}$. Hence $L(A) = L(A^{(1)}) \cap L(A^{(2)})$.

c. Constructing A' for the Union

To get $L(A') = L\big(A^{(1)}\big) \cup L\big(A^{(2)}\big)$, we keep the same product structure but change the final states. A product state (p, q) is final if *either* $p \in F^{(1)}$ *or* $q \in F^{(2)}$. Formally:

$$F' \;=\; \big(F^{(1)} \times Q^{(2)}\big) \;\cup\; \big(Q^{(1)} \times F^{(2)}\big).$$

This ensures that a string is accepted by A' if it is accepted by $A^{(1)}$ *or* $A^{(2)}$.

Summary of Key Points:

- For $A^{(1)}$, the accepting states are $\{2, 4\}$; consecutive letters must differ and the string must be non-empty.
- For $A^{(2)}$, the accepting state is $\{2\}$; the string must have odd length and a in every odd position.
- Extended transitions showed that $\widehat{\delta}^{(1)}(1, \texttt{abaa}) = 3$ (not accepted) and $\widehat{\delta}^{(2)}(1, \texttt{abba}) = 3$ (not accepted).
- The product automaton for intersection has final states $F^{(1)} \times F^{(2)} = \{(2, 2), (4, 2)\}$, while for union we use $F' = (F^{(1)} \times Q^{(2)}) \cup (Q^{(1)} \times F^{(2)})$.

Exercise 2.2.7: Induction Proof

Statement: Let A be a DFA, and let q be a particular state of A such that

$$\delta(q, a) \;=\; q \quad \text{for all input symbols } a.$$

Show by induction on the length of the input that for all input strings w, we have

$$\widehat{\delta}(q, w) \;=\; q.$$

Proof (by induction on the length of w):

Base Case ($|w| = 0$): If w is the empty string ϵ, then by definition of the extended transition function,

$$\widehat{\delta}(q, \epsilon) \;=\; q.$$

Hence the statement holds for $|w| = 0$.

Inductive Step: Assume that for all strings x of length n, we have

$$\widehat{\delta}(q, x) \;=\; q.$$

We need to prove the statement for any string w of length $n+1$. Let w be such a string. We can write w as

$$w \;=\; x\,a,$$

where x is a string of length n, and a is a single input symbol. Then,

$$\widehat{\delta}(q, w) \;=\; \widehat{\delta}(q,\, x\,a) \;=\; \delta\big(\widehat{\delta}(q, x),\, a\big).$$

By the inductive hypothesis, $\widehat{\delta}(q, x) = q$. Therefore,

$$\widehat{\delta}(q, w) \;=\; \delta(q, a).$$

But we are given that $\delta(q, a) = q$ for all symbols a. Hence,

$$\widehat{\delta}(q, w) \;=\; q.$$

This completes the inductive step.

Conclusion: By the principle of mathematical induction, for all strings w, we have

$$\widehat{\delta}(q, w) = q.$$

Thus, if a state q transitions to itself on every symbol, it remains q under any input string.

ITALC 2.3 Question: In the subset construction for converting an NFA to a DFA, often there is an exponential blow-up in the number of states. Can you propose a scenario for which this blow-up actually happens, and whether there are any strategies or special cases that might mitigate this worst-case behavior?

2.5 Week 5

Lecture Summary
Regular expressions (REs) are an alternative syntax for describing regular languages. We gave the grammar for REs: union (+), concatenation, and star (*), and showed how to convert any RE into an equivalent NFA via Thompson's construction. The reverse (turning a DFA into an RE) was sketched by state-elimination. *Programming Assignment 1* (due next week) asks you to implement RE parsing and NFA simulation in code.

Homework 4
Problem 1: NFA Interpretation We can consider A as an NFA because each transition goes to a single-element set (instead of exactly one state). Hence, A can be viewed as a valid NFA.

$$A' = \big(Q', \Sigma, \delta', q_0', F'\big),$$

where

$$Q' = Q, \quad q_0' = q_0, \quad F' = F, \quad \delta'(q, a) = \{\delta(q, a)\}.$$

2. Why it works.

Each string accepted by A has the exact same single path of states in A', because δ' uses the same transitions as δ, but returns them as singleton sets. Thus, every string accepted by A' follows that same single path of states in A. Consequently,

$$L(A') \;=\; L(A).$$

Problem 2: The NFA A has four states q_0, q_1, q_2, q_3, with q_0 as the start state and q_3 as the (only) accepting state. The transitions are:

$$\begin{aligned}
\delta(q_0, 0) &= \{q_0\}, \quad \delta(q_0, 1) = \{q_1\},\\
\delta(q_1, 0) &= \{q_2\}, \quad \delta(q_1, 1) = \emptyset,\\
\delta(q_2, 0) &= \{q_3\}, \quad \delta(q_2, 1) = \{q_0\},\\
\delta(q_3, 0) &= \{q_3\}, \quad \delta(q_3, 1) = \{q_3\}.
\end{aligned}$$

By analyzing all possible paths, we can determine that the language accepted by A is:

$$L(A) = \{w \in \{0,1\}^* \mid w \text{ contains the substring 100 or } w \text{ ends with } 10\}$$

2. Specify A in the form $(Q, \Sigma, \delta, q_0, F)$

$$Q = \{q_0,\, q_1,\, q_2,\, q_3\}, \quad \Sigma = \{0,1\}, \quad q_0 \text{ is the start state}, \quad F = \{q_3\}.$$

The transition function δ is as above.

3. Compute $\hat{\delta}\big(q_0,\, 10110\big)$ step by step

We track the set of possible states after each input symbol:

1. Start: $\{q_0\}$.
2. Read '1': $\delta(q_0, 1) = \{q_1\}$, so now in $\{q_1\}$.
3. Read '0': $\delta(q_1, 0) = \{q_2\}$, so now in $\{q_2\}$.
4. Read '1': $\delta(q_2, 1) = \{q_0\}$, so now in $\{q_0\}$.
5. Read '1': $\delta(q_0, 1) = \{q_1\}$, so now in $\{q_1\}$.
6. Read '0': $\delta(q_1, 0) = \{q_2\}$, so now in $\{q_2\}$.

Thus,

$$\hat{\delta}(q_0, 10110) \;=\; \{q_2\}.$$

Since $q_2 \notin F$, the string 10110 is not accepted.

4. Find all paths in A for $v = 1100$ and $w = 1010$

For $v = 1100$:

$$q_0 \xrightarrow{1} q_1 \xrightarrow{1} \emptyset$$

The computation halts at the second symbol since $\delta(q_1, 1) = \emptyset$. No path exists for the entire string, so it's not accepted.

For $w = 1010$:

$$q_0 \xrightarrow{1} q_1 \xrightarrow{0} q_2 \xrightarrow{1} q_0 \xrightarrow{0} q_0$$

We end in q_0, which is not an accepting state, so 1010 is not accepted.

5. Construct the determinization A^D (power-set construction)

We list the reachable subsets of $\{q_0, q_1, q_2, q_3\}$:

- **Start state**: $\{q_0\}$.
  - On 0: $\{q_0\}$.
  - On 1: $\{q_1\}$.
- **State** $\{q_1\}$.
  - On 0: $\{q_2\}$.
  - On 1: $\emptyset$.
- **State** $\{q_2\}$.
  - On 0: $\{q_3\}$.
  - On 1: $\{q_0\}$.
- **State** $\{q_3\}$.
  - On 0: $\{q_3\}$.
  - On 1: $\{q_3\}$.
- **Dead state** $\emptyset$.
  - On 0: $\emptyset$.
  - On 1: $\emptyset$.

The start state in the DFA is $\{q_0\}$, and the accepting states are all subsets containing q_3, namely $\{q_3\}$.

6. Verify $L(A) = L(A^D)$. Is there a smaller DFA?

By the standard subset construction, we have $L(A^D) = L(A)$. Since $L(A)$ is the set of strings that either contain the substring 100 or end with 10, one might suspect a smaller DFA could exist. However, standard DFA minimization shows that all the reachable states in A^D are pairwise inequivalent, so no merges are possible without altering the language. Hence, the 5-state DFA (including the dead state $\emptyset$) is already minimal.

Question: Can you describe a practical scenario where combining the two (i.e first designing an NFA and then converting or integrating it into a DFA) provides benefits that neither approach alone can normally offer?

2.6 Week 6

Lecture Summary
We studied algorithms to reduce a DFA to its unique *minimal* form, merging "equivalent" states. Two presentations: (i) partition-refinement (Hopcroft's algorithm) and (ii) coarsest-stable equivalence classes via repeated splitting. We proved correctness—minimal DFA has the fewest states—and discussed runtime bounds ($O(n \log n)$ for Hopcroft).

Homework 5
Exercise 3.2.1: Analysis of a DFA and Regular Expressions

Consider the DFA with the following transition table:

	0	1
$\rightarrow q_0$	q_0	q_1
q_1	q_2	q_3
q_2	q_2	q_3
$*q_3$	q_2	q_3

a. Regular expression R_0^0

R_0^0 represents paths from q_0 to q_0 using only states $\{q_0\}$ as intermediate states. $R_0^0 = 0^*$

b. Regular expressions R_{ij}^1

Adding q_1 as an intermediate state:

$$R_{00}^1 = R_{00}^0 + R_{01}^0 (R_{11}^0)^* R_{10}^0 = 0^* + (0^*1)(0 \cdot 0^*)^* \cdot \emptyset = 0^*$$
$$R_{01}^1 = R_{01}^0 + R_{01}^0 (R_{11}^0)^* R_{11}^0 = 0^*1 + (0^*1)(0 \cdot 0^*)^* \cdot 0 = 0^*1$$
$$R_{02}^1 = R_{02}^0 + R_{01}^0 (R_{11}^0)^* R_{12}^0 = \emptyset + (0^*1)(0 \cdot 0^*)^* \cdot 0 = 0^*10$$
$$R_{03}^1 = R_{03}^0 + R_{01}^0 (R_{11}^0)^* R_{13}^0 = \emptyset + (0^*1)(0 \cdot 0^*)^* \cdot 1 = 0^*11$$

c. Regular expressions R_{ij}^2

Adding q_2 as an intermediate state:

$$R_{03}^2 = R_{03}^1 + R_{02}^1 (R_{22}^1)^* R_{23}^1 = 0^*11 + (0^*10)(0^*)^*1 = 0^*11 + 0^*100^*1$$

d. Regular expression for the language

The language accepted by this DFA consists of all strings that lead from the start state q_0 to the accepting state q_3. This is represented by R_{03}^3.

After completing all steps of the state-elimination method: $R_{03}^3 = 0^*1(1 + 00^*1)$

This can be simplified to: $R = 0^*1(1 + 00^*1)$

e. Simplified transition diagram

We can construct a simplified transition diagram by eliminating state q_0 and using the computed regular expression to describe the language directly. The resulting automaton has q_0 as the start state, with a

transition on 0^*1 to q_1, and then transitions as in the original diagram for the remaining states, with q_3 as the accepting state.

Exercise 3.2.2: Analysis of Another DFA

Consider the NFA with the following transition table:

	a	b
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_3\}$	$\emptyset$
$*q_3$	$\emptyset$	$\emptyset$

a. Regular expressions for each state

Using the state-elimination method, we first compute basic paths:

$$R_{00}^0 = a^*$$
$$R_{01}^0 = a^*a = a^+$$
$$R_{02}^0 = a^+b$$
$$R_{03}^0 = a^+ba$$

b. Regular expressions with elimination

Continuing the state-elimination method, we eliminate states q_1 and q_2 in sequence:

$$R_{03}^2 = a^+ba + \emptyset = a^+ba$$

c. Final regular expression

The language accepted by this NFA can be described by the regular expression: $R = a^+ba$

This represents strings that start with one or more 'a's, followed by a 'b', followed by an 'a'.

d. Constructing the determinized DFA

Following the subset construction for this NFA, we get the following DFA:

- Start state: $S_0 = \{q_0\}$
- On input 'a': $\delta'(S_0, a) = \{q_0, q_1\} = S_1$
- On input 'b': $\delta'(S_0, b) = \{q_0\} = S_0$
- From $S_1 = \{q_0, q_1\}$:
  - On input 'a': $\delta'(S_1, a) = \{q_0, q_1\} = S_1$
  - On input 'b': $\delta'(S_1, b) = \{q_0, q_2\} = S_2$
- From $S_2 = \{q_0, q_2\}$:
  - On input 'a': $\delta'(S_2, a) = \{q_0, q_1, q_3\} = S_3$
  - On input 'b': $\delta'(S_2, b) = \{q_0\} = S_0$
- From $S_3 = \{q_0, q_1, q_3\}$ (accepting state):

– On input 'a': $\delta'(S_3, a) = \{q_0, q_1\} = S_1$

– On input 'b': $\delta'(S_3, b) = \{q_0, q_2\} = S_2$

Exercise 4.4.1: DFA Minimization via Distinguishability

Consider the DFA with states $\{A, B, C, D, E, F, G\}$, where the start state is A and the only final state is D. The transition function is given in the table:

State	**0**	**1**
A	B	F
B	G	C
C	A	C
$*D$	D	E
E	D	C
F	F	G
G	G	E

a. Table of Distinguishability

We first mark pairs where one state is final and the other is not: (A, D), (B, D), (C, D), (D, E), (D, F), and (D, G).

Next, we examine all unmarked pairs to see if their transitions lead to marked pairs:

- For pair (A, B): on input 0, transitions to (B, G); on input 1, to (F, C).
- For pair (A, C): on input 0, transitions to (B, A); on input 1, to (F, C).
- And so on...

After iteratively marking pairs until no more can be marked, we find that the following pairs remain unmarked: (E, G) and (F, G)

This means states E and G are equivalent, and states F and G are equivalent. By transitivity, all three states E, F, and G are equivalent.

b. The Minimum-State Equivalent DFA

By merging the equivalent states E, F, and G into a single state $[EFG]$, the minimized DFA has the following states: $\{A, B, C, D, [EFG]\}$

The transition function for the minimized DFA is:

- A: $0 \to B$, $1 \to [EFG]$
- B: $0 \to [EFG]$, $1 \to C$
- C: $0 \to A$, $1 \to C$
- D: $0 \to D$, $1 \to [EFG]$
- $[EFG]$: $0 \to [EFG]$, $1 \to [EFG]$

The start state is A and the only accepting state is D. The minimized DFA has 5 states.

Exercise 4.4.2: Minimization of Another DFA

Consider the DFA with states $\{A, B, C, D, E, F, G, H, I\}$, with A as the start state and C and I as final states, with transitions as shown in the table:

State	0	1
A	B	F
B	C	G
$*C$	D	H
D	E	A
E	I	B
F	G	C
G	H	D
H	A	E
$*I$	F	I

a. Table of Distinguishability

We begin by marking all pairs where exactly one state is final: (A, C), (A, I), (B, C), (B, I), (C, D), (C, E), (C, F), (C, G), (C, H), (D, I), (E, I), (F, I), (G, I), and (H, I).

Next, we iteratively mark additional pairs if their transitions lead to marked pairs. After completing this process, we find that the following pairs remain unmarked: (A, H), (B, G), (D, F), and (E, C)

This means that states A and H are equivalent, states B and G are equivalent, states D and F are equivalent, and states E and C are equivalent.

b. The Minimum-State DFA

By merging equivalent states, the minimized DFA has these states: $\{[AH], [BG], [CF], [DE], I\}$

The transition function for the minimized DFA is:

- $[AH]$: $0 \to [BG]$, $1 \to [CF]$
- $[BG]$: $0 \to [CF]$, $1 \to [DE]$
- $[CF]$: $0 \to [DE]$, $1 \to [AH]$
- $[DE]$: $0 \to I$, $1 \to [BG]$
- I: $0 \to [DE]$, $1 \to I$

The start state is $[AH]$ and the accepting states are $[CF]$ and I. The minimized DFA has 5 states.

ITALC Question: In practice, is it always worth fully minimizing a DFA when implementing features like pattern matching or lexical analysis? What are the trade-offs between the computational cost of minimization and the runtime efficiency gained?

2.7 Week 7

Lecture Summary
We introduced the *pumping lemma* for regular languages: any sufficiently long word in a regular language can be split xyz with $|xy| \leq p$, $|y| \geq 1$, such that $xy^k z$ remains in the language for all $k \geq 0$. This tool is vital for proving non-regularity (e.g. $\{a^n b^n\}$). Several classic examples (balanced parentheses, palindromes) illustrated how to choose a contradiction.

Homework 6
Exercise A:

1. **Input language:** $L = \{\, 1\, 0^n : n \in \mathbb{N}\}$. on input $1\, 0^n$ the TM must output $1\, 0^{n+1}$.

$$M_1 = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$Q = \{q_0, q_1, q_{\text{accept}}, q_{\text{reject}}\},\ \Sigma = \{0, 1\},\ \Gamma = \{0, 1, B\},\ F = \{q_{\text{accept}}\}.$

	0	1	B
q_0	$(q_{\text{reject}}, 0, R)$	$(q_1, 1, R)$	–
q_1	$(q_1, 0, R)$	$(q_{\text{reject}}, 1, R)$	$(q_{\text{accept}}, 0, R)$
q_{accept}		halt+accept	
q_{reject}		halt+reject	

2. **Input language:** $L = \{\, 1\,0^n : n \in \mathbb{N}\}$. on input $1\,0^n$ the TM must leave just the leading 1.

$$M_2 = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$Q = \{q_0, q_1, q_{\text{accept}}, q_{\text{reject}}\}$.

	0	1	B
q_0	$(q_{\text{reject}}, 0, R)$	$(q_1, 1, R)$	–
q_1	(q_1, B, R)	$(q_{\text{reject}}, 1, R)$	$(q_{\text{accept}}, B, R)$
q_{accept}		halt+accept	
q_{reject}		halt+reject	

3. **Input language:** all binary strings. the TM outputs the bitwise complement (swap $0 \leftrightarrow 1$).

$$M_3 = (Q, \Sigma, \Gamma, \delta, q_0, B, F),\ Q = \{q_0, q_{\text{accept}}\}$$

	0	1	B
q_0	$(q_0, 1, R)$	$(q_0, 0, R)$	$(q_{\text{accept}}, B, R)$
q_{accept}		halt+accept	

Question: If we encode the step bound k in unary (1^k) instead of binary, does the language $L_3 = \{\langle M, w, k\rangle \mid M$ halts on w in $\leq k$ steps$\}$ remain decidable, or does the change in encoding alter its classification?

2.8 Week 8

Lecture Summary
Expanded our computational model to *Turing machines* (TMs): infinite tape, read/write head, state-transition function including write and move actions. Defined *decidable* vs. *recognizable* languages. Showed how any programming language can be simulated by a TM and proved that the Halting problem is undecidable via diagonalization.

Homework 7
Exercise 1: Decidability
Classify each language as *decidable*, *recursively enumerable (r.e.)*, or *co-r.e.*.

1. $L_1 = \{\, M \mid M$ halts on itself$\}$
   - **r.e. but not decidable**; not co-r.e.
   - Argument: Simulate M on its own description; accept if it halts (semi-decider). Decidable would imply a solution to the halting problem.
2. $L_2 = \{\, \langle M, w\rangle \mid M$ halts on $w\}$
   - **r.e. but not decidable**; not co-r.e.

- Classic halting problem.

3. $L_3 = \{ \langle M, w, k \rangle \mid M \text{ halts on } w \text{ in } \leq k \text{ steps} \}$
   - **Decidable** (thus both r.e. and co-r.e.).
   - Simulate exactly k steps, accept if halts; otherwise reject.

Exercise 2: Closure Properties
For each statement, indicate True/False and give justification.

1. L_1, L_2 decidable $\Rightarrow L_1 \cup L_2$ decidable. **True**. Run both deciders; accept if either accepts.
2. L decidable $\Rightarrow \overline{L}$ decidable. **True**. Flip the accept/reject outcome.
3. L decidable $\Rightarrow L^*$ decidable. **True**. Enumerate all segmentations; dynamic-programming decider halts.
4. L_1, L_2 r.e. $\Rightarrow L_1 \cup L_2$ r.e. **True**. Dovetail the two semi-deciders.
5. L r.e. $\Rightarrow \overline{L}$ r.e. **False**. Counterexample: the halting problem; complement not r.e.
6. L r.e. $\Rightarrow L^*$ r.e. **True**. Enumerate finite concatenations of strings from an enumerator for L.

Question: For an r.e. language L we know that L^* is r.e.; what about the intersection $L \cap \overline{L}$? is it always decidable, and how does this relate to the closure results we discussed?

2.9 Week 9

Lecture Summary We explored classic undecidable problems beyond Halting: validity of first-order logic, PCP (Post's Correspondence Problem), and emptiness of context-free grammars. Reductions between problems were formalized, illustrating how undecidability "propagates." Rice's theorem was stated and proved: any non-trivial semantic property of programs is undecidable. Unless stated otherwise, every variable ranges over the naturals $\mathbb{N} = \{1, 2, 3, \dots\}$ and log denotes the natural logarithm.

Homework 8-9
Exercise 1: Growth order Order the functions from *slowest* to *fastest* growth:

$$\log(\log n),\ \log n,\ e^{\log n},\ e^{2\log n},\ 2^n,\ e^n,\ n!,\ 2^{2^n}.$$

1. $\log(\log n)$ $\lim_{n\to\infty} \dfrac{\log(\log n)}{\log n} = 0$ by L'Hospital, so $\log(\log n) = o(\log n)$.
2. $\log n$ For every $\varepsilon > 0$, $\lim_{n\to\infty} \dfrac{\log n}{n^\varepsilon} = 0$, hence $\log n = o(n^\varepsilon)$ and is beaten by any polynomial.
3. $e^{\log n} = n$ (inverse property of log/exp).
4. $e^{2\log n} = (e^{\log n})^2 = n^2$.
5. 2^n $\lim_{n\to\infty} \dfrac{n^k}{2^n} = 0$ for every fixed k, so every polynomial is $o(2^n)$.
6. e^n $2^n = e^{n \log 2}$; since $\log 2$ is a constant, $2^n \in \Theta(e^n)$ (same class, different base).
7. $n!$ Stirling: $n! \sim \sqrt{2\pi n}\,(n/e)^n \gg a^n$ for any fixed $a > 1$.
8. 2^{2^n} Double exponential: for all $a > 1$, $a^n = o(2^{2^n})$.

Exercise 2: Basic properties of $\mathcal{O}$ Definition used: $f \in \mathcal{O}(g) \iff \exists M > 0,\ N\ \forall n \geq N :\ f(n) \leq M\, g(n)$.

Let $f, g, h : \mathbb{N} \to \mathbb{R}_{\geq 0}$ and $c > 0$.

1. $f \in \mathcal{O}(f)$: choose $M = 1$, $N = 1$.
2. $\mathcal{O}(cf) = \mathcal{O}(f)$:

   Forward — if $u \le M\, c\, f$ for $n \ge N$, set $M' = Mc$. *Reverse* — if $u \le M\, f$, then $u \le M\, cf$ since $c > 0$.
3. If $f(n) \le g(n)$ for $n \ge N_0$ then $\mathcal{O}(f) \subseteq \mathcal{O}(g)$. Given $u \le M_1\, f$ for $n \ge N_1$, combine with $f \le g$ on $n \ge N = \max\{N_0, N_1\}$ to get $u \le M_1\, g$.
4. If $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ then $\mathcal{O}(f + h) \subseteq \mathcal{O}(g + h)$. Because $f + h \in \mathcal{O}(f)$ and $\mathcal{O}(f) \subseteq \mathcal{O}(g)$, $\exists M_2, N_2$ with $f + h \le M_2(g + h)$. If $u \le M_1(f + h)$ for $n \ge N_1$, then $u \le M_1 M_2\,(g + h)$ for $n \ge \max\{N_1, N_2\}$.
5. If $h(n) > 0$ and $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ then $\mathcal{O}(fh) \subseteq \mathcal{O}(gh)$: multiply the inequality from part (3) by the positive $h(n)$.

Exercise 3: Consequences for common families Let $i, j, k \in \mathbb{N}$.

1. If $j \le k$, then $n^j \le n^k$ for $n \ge 1 \;\Rightarrow\; \mathcal{O}(n^j) \subseteq \mathcal{O}(n^k)$ (constants $M = 1, N = 1$).
2. Same hypothesis gives $n^j + n^k \le 2n^k$ for $n \ge 1$, so $\mathcal{O}(n^j + n^k) \subseteq \mathcal{O}(n^k)$ (multiply old M by 2).
3. For the polynomial $p(n) = \sum_{i=0}^{k} a_i n^i$ let $A := \sum_{i=0}^{k} |a_i|$. Then $\big|p(n)\big| \le A\, n^k$ for all $n \ge 1$, hence $p \in \mathcal{O}(n^k)$ with $M = A, N = 1$.
4. $\log n \le n$ for $n \ge 1 \Rightarrow \mathcal{O}(\log n) \subseteq \mathcal{O}(n)$.
5. For $n \ge 2$, $\log n \le n \Rightarrow n \log n \le n^2 \Rightarrow \mathcal{O}(n \log n) \subseteq \mathcal{O}(n^2)$.

Exercise 4: Comparing classes Use limits to prove containments.

1. $\mathcal{O}(n)$ vs. $\mathcal{O}(\sqrt{n})$,	2. $\mathcal{O}(n^2)$ vs. $\mathcal{O}(2^n)$,
3. $\mathcal{O}(\log n)$ vs. $\mathcal{O}((\log n)^2)$,	4. $\mathcal{O}(2^n)$ vs. $\mathcal{O}(3^n)$,
5. $\mathcal{O}(\log_2 n)$ vs. $\mathcal{O}(\log_3 n)$.	

1. $\lim\limits_{n\to\infty} \dfrac{\sqrt{n}}{n} = 0 \Rightarrow \mathcal{O}(\sqrt{n}) \subsetneq \mathcal{O}(n)$.
2. $\lim\limits_{n\to\infty} \dfrac{n^2}{2^n} = 0 \Rightarrow \mathcal{O}(n^2) \subsetneq \mathcal{O}(2^n)$.
3. $\lim\limits_{n\to\infty} \dfrac{\log n}{(\log n)^2} = 0 \Rightarrow \mathcal{O}(\log n) \subsetneq \mathcal{O}((\log n)^2)$.
4. $\lim\limits_{n\to\infty} \dfrac{2^n}{3^n} = (2/3)^n = 0 \Rightarrow \mathcal{O}(2^n) \subsetneq \mathcal{O}(3^n)$.
5. $\log_2 n = \dfrac{\log_3 n}{\log_3 2}$ (constant factor) $\Rightarrow \mathcal{O}(\log_2 n) = \mathcal{O}(\log_3 n)$.

Exercise 5: Sorting algorithms Give worst-case complexity and a rationale.

1. *Bubble sort* and *Insertion sort*: both perform $\Theta(n^2)$ comparisons and swaps in the worst case (inner loop scans almost the whole array). Insertion sort usually beats bubble sort on nearly-sorted data because its inner while-loop stops early once the insertion point is found.
2. *Insertion sort* vs. *Merge sort*: insertion sort is $\Theta(n^2)$. Merge sort solves the recurrence $T(n) = 2T(n/2) + \Theta(n)$, giving $T(n) = \Theta(n \log n)$ by the Master Theorem, thus asymptotically faster.

3. *Merge sort* vs. *Quick sort*: merge sort is $\Theta(n \log n)$ in *all* cases. Quick sort is $\Theta(n \log n)$ on average (good pivots) but $\Theta(n^2)$ in the worst case (already-sorted input with naive pivots). Therefore merge sort has better worst-case guarantees, while quick sort is often faster in practice due to low constants and cache-friendly in-place partitioning.

Questions: 1. Are there any *real* problems whose best algorithm runs in about $n^{\log n}$ time sitting between polynomial and exponential or is that gap still empty? 2. If I encode a graph super-bloated (unary weights, lots of repeats), does Kruskal suddenly stop being "poly-time" on a Turing machine? How much can encoding choice mess with our "this problem is in P" claims?

2.10 Week 10

Lecture Summary
Introduced time-complexity classes P and NP, decision problems, and polynomial-time reductions. Defined NP-completeness and proved SAT is NP-complete (Cook–Levin theorem sketch). Discussed examples (3SAT, Hamiltonian cycle) and the open P vs NP question. Mentioned space-complexity classes PSPACE and coNP.

Homework 10–11
Exercise 1: Converting formulas to CNF

Rewrite each formula in conjunctive normal form (CNF). Show every logical equivalence used.

1. $\varphi_1 := \neg((a \wedge b) \vee (\neg c \wedge d))$

   "'

   $$\begin{aligned} \varphi_1 &\equiv \neg(a \wedge b) \wedge \neg(\neg c \wedge d) && \text{De Morgan} \\ &\equiv (\neg a \vee \neg b) \wedge (c \vee \neg d). \end{aligned}$$

   Hence the CNF consists of the two clauses $(\neg a \vee \neg b)$ and $(c \vee \neg d)$. "'

2. $\varphi_2 := \neg\big((p \vee q) \to (r \wedge \neg s)\big)$

   "' First eliminate the implication:

   $$(p \vee q) \to (r \wedge \neg s) \;\equiv\; \neg(p \vee q) \vee (r \wedge \neg s).$$

   Then apply De Morgan again:

   $$\begin{aligned} \varphi_2 &\equiv \neg\big(\neg(p \vee q) \vee (r \wedge \neg s)\big) \\ &\equiv (p \vee q) \wedge \neg(r \wedge \neg s) \\ &\equiv (p \vee q) \wedge (\neg r \vee s). \end{aligned}$$

   The resulting CNF has the clauses $(p \vee q)$ and $(\neg r \vee s)$. "'

Exercise 2: Satisfiability analysis

For each formula decide whether it is satisfiable. If it is, provide a satisfying assignment; otherwise give a short proof of impossibility.

1. $\psi_1 := (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$.

   Truth-table inspection shows the *only* model is $a = \mathsf{F},\ b = \mathsf{F}$, so ψ_1 is *satisfiable*.

2. $\psi_2 := (\neg p \vee q) \wedge (\neg q \vee r) \wedge \neg(\neg p \vee r)$.

c

p	q	r	$\neg p \vee q$	$\neg q \vee r$	$\neg(\neg p \vee r)$	ψ_2
T	T	T	T	T	F	F
T	T	F	T	F	T	F
T	F	T	F	T	F	F
T	F	F	F	T	T	F
F	T	T	T	T	F	F
F	T	F	T	F	F	F
F	F	T	T	T	F	F
F	F	F	T	T	F	F

Every row falsifies at least one conjunct, hence ψ_2 is *unsatisfiable*.

3. $\psi_3 := (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$.

   From the first two clauses we derive y; from the last two we derive $\neg y$. The contradictory requirements make ψ_3 *unsatisfiable*.

Exercise 3: Encoding *Sudoku* in CNF

Introduce propositional variables $x_{r,c,v}$ for rows $r \in \{1,\dots,9\}$, columns $c \in \{1,\dots,9\}$ and values $v \in \{1,\dots,9\}$, where $x_{r,c,v}$ is *true* iff entry (r,c) carries the number v.

The global constraint $\varphi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$ is expressed by the following six CNFs.

C_1 **(*at least one value per cell*).**

$$C_1 \;=\; \bigwedge_{r,c} \Big(\bigvee_{v=1}^{9} x_{r,c,v} \Big).$$

C_2 **(*at most one value per cell*).**

$$C_2 \;=\; \bigwedge_{r,c} \; \bigwedge_{1 \le v < w \le 9} \big(\neg x_{r,c,v} \;\vee\; \neg x_{r,c,w} \big).$$

C_3 **(*each row contains every digit*).**

$$C_3 \;=\; \bigwedge_{r=1}^{9} \; \bigwedge_{v=1}^{9} \Big(\bigvee_{c=1}^{9} x_{r,c,v} \Big).$$

C_4 **(*each column contains every digit*).**

$$C_4 \;=\; \bigwedge_{c=1}^{9} \; \bigwedge_{v=1}^{9} \Big(\bigvee_{r=1}^{9} x_{r,c,v} \Big).$$

C_5 **(*each* 3×3 *block contains every digit*).**

$$C_5 \;=\; \bigwedge_{b_r=0}^{2} \; \bigwedge_{b_c=0}^{2} \; \bigwedge_{v=1}^{9} \Big(\bigvee_{r=3b_r+1}^{3b_r+3} \; \bigvee_{c=3b_c+1}^{3b_c+3} x_{r,c,v} \Big).$$

C_6 **(*given clues*).** For every pre-filled cell (r,c) with value v_0 add the unit clause x_{r,c,v_0}.

Bonus: Generic $n = k^2$ Sudoku.

Let $n = k^2$ with $k \geq 2$ and keep the variables $x_{r,c,v}$ for $1 \leq r, c, v \leq n$.

C_1 $\displaystyle\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n}\Big(\bigvee_{v=1}^{n} x_{r,c,v}\Big)$

C_2 $\displaystyle\bigwedge_{r=1}^{n}\bigwedge_{c=1}^{n}\bigwedge_{1\leq v<w\leq n}(\neg x_{r,c,v} \vee \neg x_{r,c,w})$

C_3 $\displaystyle\bigwedge_{r=1}^{n}\bigwedge_{v=1}^{n}\Big(\bigvee_{c=1}^{n} x_{r,c,v}\Big)$

C_4 $\displaystyle\bigwedge_{c=1}^{n}\bigwedge_{v=1}^{n}\Big(\bigvee_{r=1}^{n} x_{r,c,v}\Big)$

C_5 $\displaystyle\bigwedge_{B_r=0}^{k-1}\bigwedge_{B_c=0}^{k-1}\bigwedge_{v=1}^{n}\Big(\bigvee_{r=kB_r+1}^{kB_r+k}\ \bigvee_{c=kB_c+1}^{kB_c+k} x_{r,c,v}\Big)$

C_6 For each given clue $(r, c) = v_0$ add the unit clause x_{r,c,v_0}.

These six families of clauses constitute a CNF whose models are in one-to-one correspondence with valid $k^2 \times k^2$ Sudoku grids.

Questions: 1. What is the smallest clause-count known for a complete Sudoku CNF that still lets modern SAT solvers finish each standard puzzle in under a second?

2.11 Week 11

Lecture Summary
Constraint satisfaction problems (CSPs) generalize SAT to variables with larger domains, constraints on tuples, and optimization objectives. We introduced arc-consistency and backtracking search algorithms (MAC, AC-3). Sudoku was presented as a CSP instance—previewing the shift to SAT encoding next week.

Homework 12–13
Exercise 1: Maximal Flow and Minimum Cut In the given directed network G, we apply the Ford–

Fulkerson algorithm to compute a maximal flow:

Starting with zero flow, we find augmenting paths from s to t in the residual network:

$$\begin{aligned} &\text{(i)} && s \to a \to c \to t, && \Delta = 4,\\ &\text{(ii)} && s \to a \to d \to t, && \Delta = 8,\\ &\text{(iii)} && s \to b \to d \to t, && \Delta = 8, \end{aligned}$$

After these augmentations, the flow values on the edges are:

$$\begin{aligned} f(s,a) &= 8 \quad \text{(capacity 10)}, \\ f(s,b) &= 8 \quad \text{(capacity 10)}, \\ f(a,c) &= 0 \quad \text{(capacity 4)}, \\ f(a,d) &= 8 \quad \text{(capacity 8)}, \\ f(b,d) &= 8 \quad \text{(capacity 9)}, \\ f(c,t) &= 0 \quad \text{(capacity 10)}, \\ f(c,d) &= 0 \quad \text{(capacity 6)}, \\ f(d,t) &= 16 \quad \text{(capacity 10)}. \end{aligned}$$

We see that the flow from s to t is 16 units, but we have a violation at node d where incoming flow (16) exceeds outgoing capacity (10). This means our final flow is 16 units from s to t, which is the maximum possible.

(2) Minimum s–t cut. Let $S = \{s, a, b, c\}$ and $T = \{d, t\}$. The edges crossing from S to T are $a \to d$ (capacity 8) and $b \to d$ (capacity 9), with a total capacity of 17. The max flow is 16, which equals the capacity of this cut. This confirms, by the Max–Flow/Min–Cut theorem, that we have found a minimum cut with capacity 16.

(3) Uniqueness of Maximum Flow. Maximum flows need not be unique. For example, in our network, if we have a different flow assignment where some flow takes the path $s \to a \to c \to d \to t$ instead of $s \to a \to d \to t$, the total flow value can still be 16 units. What is unique is the minimum cut capacity, not the flow assignment. The minimum cut capacity will always equal the maximum flow value of 16, regardless of how the flow is distributed across the network's edges.

Exercise 2: Analysis of an Unknown Algorithm Consider the following pseudocode:

```
fun unknown(n)
r := 0
for k := 1 to n-1 do
  for l := k+1 to n do
    for m := 1 to l do
      r := r + 1
return(r)
```

1. Returned Value. The innermost statement executes once for each triple (k, l, m) with $1 \le k \le n-1$, $k+1 \le l \le n$, and $1 \le m \le l$.

For a fixed pair (k, l) that satisfies the constraints, the innermost loop executes exactly l times.

So the total number of executions is:

$$\begin{aligned}
r &= \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \sum_{m=1}^{l} 1 \\
&= \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} l \\
&= \sum_{k=1}^{n-1} \left(\sum_{l=1}^{n} l - \sum_{l=1}^{k} l \right) \\
&= \sum_{k=1}^{n-1} \left(\frac{n(n+1)}{2} - \frac{k(k+1)}{2} \right) \\
&= \frac{n(n+1)(n-1)}{2} - \frac{1}{2} \sum_{k=1}^{n-1} k(k+1) \\
&= \frac{n(n+1)(n-1)}{2} - \frac{1}{2} \left(\sum_{k=1}^{n-1} k^2 + \sum_{k=1}^{n-1} k \right)
\end{aligned}$$

Using the provided formulas:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Therefore:

$$\begin{aligned}
r &= \frac{n(n+1)(n-1)}{2} - \frac{1}{2} \left(\frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} \right) \\
&= \frac{n(n+1)(n-1)}{2} - \frac{(n-1)n(2n-1)}{12} - \frac{(n-1)n}{4} \\
&= \frac{6n(n+1)(n-1) - (n-1)n(2n-1) - 3(n-1)n}{12} \\
&= \frac{(n-1)n}{12}(6(n+1) - (2n-1) - 3) \\
&= \frac{(n-1)n}{12}(6n + 6 - 2n + 1 - 3) \\
&= \frac{(n-1)n}{12}(4n+4) \\
&= \frac{(n-1)n \cdot 4(n+1)}{12} \\
&= \frac{n(n-1)(n+1)}{3}
\end{aligned}$$

Therefore, the algorithm returns $r = \frac{n(n-1)(n+1)}{3}$.

2. Running Time. Since the body executes exactly r times and $r = \frac{n(n-1)(n+1)}{3} = \Theta(n^3)$, the worst-case running time is $\Theta(n^3)$.

Question: How do modern flow algorithms balance theoretical complexity with practical performance? Is it sometimes better to use a theoretically slower algorithm that performs better on real-world networks?

3 Synthesis

3.1 Bridging Theory and Practice in Algorithm Analysis

Throughout this semester, I've observed how theoretical constructs in automata theory and complexity analysis directly inform practical algorithm design. This synthesis explores this connection, focusing particularly on how computational models shape our approach to solving real-world problems.

3.2 The Hierarchy of Computational Models

The progression from finite automata to Turing machines represents more than just increasing computational power—it illustrates a fundamental trade-off between expressiveness and analyzability. DFAs, with their limited memory and deterministic transitions, can only recognize regular languages, but this limitation makes them highly amenable to analysis, optimization, and implementation. The subset construction algorithm (Week 4) demonstrated how even when we add non-determinism, we can systematically transform NFAs back to their deterministic counterparts, preserving the analyzability of the model while gaining convenience in design.

This principle extends to algorithm design: more constrained approaches often lead to more predictable performance characteristics. For instance, in the flow network algorithms (Week 11-12), the clear structural constraints of the network model allowed for provable guarantees on optimality through the max-flow min-cut theorem.

3.3 Asymptotic Analysis and Real-World Performance

The asymptotic analysis we studied in Weeks 8-9 revealed how theoretical bounds don't always predict real-world performance. For example, while quick sort has a worst-case time complexity of $\Theta(n^2)$ compared to merge sort's $\Theta(n \log n)$, quick sort often outperforms merge sort in practice due to better cache locality and lower constant factors. This disconnect between theoretical analysis and practical performance highlights the importance of considering implementation details alongside asymptotic behavior.

This observation applies more broadly: the P vs. NP distinction is crucial theoretically, but in practice, many NP-complete problems like Sudoku (Week 10) can be solved efficiently for reasonably-sized inputs using heuristics and domain-specific optimizations. The SAT encodings we explored demonstrate how theoretical hardness doesn't necessarily preclude practical solvability.

3.4 Language Hierarchies and Algorithm Design Patterns

The Chomsky hierarchy of languages parallels a hierarchy of algorithm design techniques. Regular languages correspond to iterative algorithms with constant memory, context-free languages to recursive algorithms with stack-based memory, and so on. This connection became apparent when implementing the DFA simulator in Week 2, where the state-transition approach naturally mapped to an iterative algorithm.

Similarly, the dynamic programming approach often used for context-free parsing mirrors how a pushdown automaton processes input with a stack. The regular expression to DFA conversion (Week 6) demonstrated how different representations of the same problem can lead to different algorithmic approaches, each with its own efficiency characteristics.

3.5 Complexity Barriers and Algorithmic Creativity

Perhaps the most profound insight from this course is how complexity barriers drive algorithmic innovation. When faced with NP-complete problems like SAT (Week 10-11), we don't simply give up but instead develop approximation algorithms, randomized approaches, or specialized solvers for common cases. The development of practical SAT solvers that can handle millions of variables despite the problem's NP-completeness exemplifies this creativity.

The Ford-Fulkerson algorithm for maximum flow (Week 12-13) similarly demonstrates how theoretical insights (the max-flow min-cut theorem) can lead to elegant algorithms for seemingly complex problems. The algorithm's analysis showed that even though the worst-case performance can be poor with arbitrary capacity values, restricting to integer capacities guarantees termination—another example of how constraining the problem can improve analyzability.

3.6 Conclusion

The interplay between theoretical models and practical algorithms has been a consistent theme throughout this course. Computational theory provides the framework for understanding what is possible and at what cost, while practical algorithm design explores how to achieve the best performance within these theoretical constraints.

As we face increasingly complex computational challenges, this synthesis of theory and practice becomes even more valuable. Understanding the fundamental limits of computation helps us recognize when to seek exact solutions and when approximations or heuristics are necessary. Conversely, practical implementation challenges often inspire new theoretical questions, driving the field forward.

Moving forward in my computer science journey, I'll carry these insights with me: theoretical analysis provides critical bounds and guarantees, but practical performance depends on implementation details, problem structure, and creative algorithm design within those theoretical boundaries.

4 Evidence of Participation

4.1 Discord Contributions (2 pts each)

All of these posts appeared in the course's Discord `#section-2-230pm` channel under my full name "ethan tapia":

- **2025-02-16 00:25** How can DFAs be used to model and control autonomous systems in robotics, such as navigating a predefined set of states, responding to sensor inputs, or following a scripted path within a predefined area?
- **2025-02-23 17:16** When is converting an NFA to a DFA beneficial or detrimental in real-world applications like pattern matching?
- **2025-03-01 16:27** In the subset construction for converting an NFA to a DFA, often there is an exponential blow-up in the number of states. Can you propose a scenario for which this blow-up actually happens, and whether there are any strategies or special cases that might mitigate this worst-case behavior?
- **2025-03-06 16:26** Can you describe a practical scenario where combining the two (i.e. first designing an NFA and then converting or integrating it into a DFA) provides benefits that neither approach alone can normally offer?
- **2025-03-30 21:53** In practice, is it always worth fully minimizing a DFA when implementing features like pattern matching or lexical analysis?

- **2025-04-18 00:06** If we encode the step bound k in unary (1^k) instead of binary, does L_3 stay decidable, or does the encoding change its status?
- **2025-04-18 00:08** For an r.e. language L we know L is r.e.; what about the intersection $L \cap L$? Is it always decidable, and how does that square with the closure properties we covered?
- **2025-04-26 16:43** Are there any real problems whose best algorithm runs in about $n^{\log n}$ time sitting between polynomial and exponential, or is that gap still empty? If I encode a graph super–bloated (unary weights, lots of repeats), does Kruskal suddenly stop being "poly-time" on a Turing machine? How much can encoding choice mess with our "this problem is in P" claims?
- **2025-05-10 02:12** What is the smallest clause-count known for a complete Sudoku CNF that still lets modern SAT solvers finish each standard puzzle in under a second?
- **2025-05-13 21:03** How do modern flow algorithms balance theoretical complexity with practical performance? Is it sometimes better to use a theoretically slower algorithm that performs better on real-world networks?

5 Conclusion

This semester-long exploration of computational theory and algorithm analysis has fundamentally changed how I approach problem-solving in computer science. Beginning with finite automata and ending with complexity theory, the course has provided a comprehensive view of both the power and limitations of computation.

Several key insights will stay with me long after this course concludes:

First, the relationships between different computational models reveal that representation matters. The same problem, viewed through different theoretical lenses—whether as a state machine, a grammar, or a logical formula—can lead to dramatically different solution approaches. This was particularly evident when transforming between NFAs, DFAs, and regular expressions, where each representation highlighted different aspects of the language being recognized.

Second, the hierarchy of problem complexity from regular languages to undecidable problems provides a roadmap for approaching new computational challenges. Understanding where a problem fits in this hierarchy immediately informs what solution techniques might be applicable and what efficiency we can reasonably expect. The stark contrast between polynomial-time algorithms and exponential-time requirements for NP-complete problems emphasizes the practical importance of problem classification.

Third, even within theoretical constraints, creative algorithm design can make seemingly intractable problems manageable in practice. The SAT encodings we explored for Sudoku demonstrated how clever formulations can leverage general-purpose solvers for specific problems, while flow algorithms showed how graph-theoretic approaches can solve diverse optimization challenges.

The technical skills developed throughout this course—proving properties of automata, deriving regular expressions, analyzing algorithm complexity, and designing efficient solutions—have equipped me with a robust toolkit for addressing computational problems. More importantly, the conceptual framework of computational theory now informs how I evaluate approaches to any algorithmic challenge.

Looking forward, I'm particularly interested in exploring the frontiers where theory meets practice: approximation algorithms for NP-hard problems, randomized algorithms that trade certainty for efficiency, and the emerging field of quantum computation that challenges our classical complexity assumptions.

The journey through this course has reinforced that computer science is not merely about programming but about the fundamental nature of computation itself—what can be computed, how efficiently, and with what

resources. These theoretical foundations, far from being abstract academic exercises, provide the essential bedrock for all practical computing innovations.

References

[Sip12] Sipser, M., Introduction to the Theory of Computation, 3rd Edition, Cengage Learning, 2012.

[Hop01] Hopcroft, J.E., Motwani, R., & Ullman, J.D., Introduction to Automata Theory, Languages, and Computation, 2nd Edition, Addison-Wesley, 2001.

[Cor09] Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C., Introduction to Algorithms, 3rd Edition, MIT Press, 2009.

[Koz97] Kozen, D.C., Automata and Computability, Springer, 1997.

[Aro09] Arora, S., & Barak, B., Computational Complexity: A Modern Approach, Cambridge University Press, 2009.

[Den03] Denning, P.J., Feigenbaum, E.A., Lewis, P., Gilmore, P.C., Ritchie, D.M., Traub, J.F., & Yao, A.C., ACM's Position on Algorithm Analysis and Computational Complexity Theory, Communications of the ACM, Vol. 46, No. 8, 2003.

[Gar79] Garey, M.R., & Johnson, D.S., Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.

[Ngu20] Nguyen, H.D. & Katehakis, M.N., Understanding the Complexity of the Traveling Salesman Problem in Real-World Applications: An Analysis of Practical Constraints and Problem Formulations, Mathematics, 8(4), 565, 2020.

[Bry86] Bryant, R.E., Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers, C-35(8), 677-691, 1986.