CPSC-406 Report

Abstract Machines and Formal Languages

Max Randall
Chapman University

May 25, 2025

Abstract

Automata are mathematical models of computation capturing devices that operate without unbounded memory. In this report we introduce the basic definitions of deterministic and nondeterministic finite automata through illustrative examples, and outline their role in defining and recognizing regular languages.

Contents

1 Introduction

This report documents my learning journey in CPSC-406: Algorithm Analysis, compiling each week's lecture notes, homework solutions, exploratory reflections, and questions on topics ranging from finite automata and formal languages through Turing machines and decidability to NP-completeness and graph-theoretic algorithms. Each weekly entry is organized in the order of Notes–Homework–Exploration–Questions, ensuring consistent organization of definitions, proofs, code implementations, and conceptual insights. The report concludes with dedicated Synthesis, Evidence of Participation, and Conclusion sections that tie together the semester's themes, and reflect on the relavence of the aformentioned concepts.

2 Week by Week

2.1 Week 1: Finite Automata Fundamentals

2.1.1 Introduction

Automata are abstract machines that model computations without memory. Before defining them formally, we consider some examples.

2.1.2 Readings

Parking or Vending Machine **Specification:** The machine requires 25 cents, paid in chunks of 5 or 10 cents.

Automaton: The state 25 is the accepting or final state. A word (i.e., a sequence of symbols 5 and 10) is accepted if it leads from the initial state (0) to the final state (25).

$$\begin{tikzcd}
\text{start} \arrow[r] & \text{0¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{5¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{10¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{15¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{20¢} \arrow[r, bend left, "\text{5¢}"] & \text{25¢}
\end{tikzcd}$$

Variable Names **Specification:** In defining a programming language, valid variable names should:

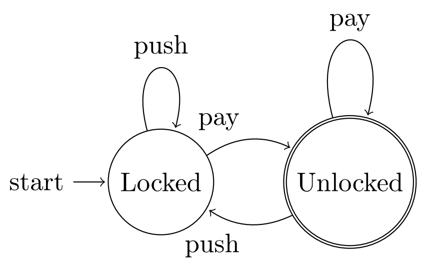
- Start with a letter ($\ell = a, b, c, \ldots, z$).
- Be followed by any combination of letters (ℓ) or digits ($d = 0, 1, 2, \ldots, 9$).
- End with a terminal symbol (t, e.g., $t =;$).

Automaton: Accepted words follow the pattern:

$$\ell(\ell + d)^* t$$

Turnstile Specification: A money-operated turnstile:

- Starts in the **locked** state.
- From **locked**:
  - A **push** (u) keeps it **locked**.
  - A **pay** (p) moves it to the **unlocked** state.
- From **unlocked**:
  - A **pay** (p) keeps it **unlocked**.
  - A **push** (u) moves it back to **locked**.
- The **unlocked** state is the accepting state.



2.1.3 Homework

Exercise: Characterizing Accepted Words Characterize all accepted words (i.e., describe exactly those words that are recognized).

The vending machine automaton accepts words consisting of payments in increments of 5 and 10 cents, reaching exactly 25 cents. The accepted words follow the pattern:

$$(10 + 5)^*$$

subject to the constraint that the sum of the numbers in the sequence equals 25.

Exercise: Turnstile Regular Expression Characterize all accepted words and describe them using a regular expression.

The turnstile automaton can be characterized by the following regular expression:

$$(p + up)^* p (p + up)^*$$

where:

- p represents a **pay** action.
- u represents a **push** action.
- p^* means zero or more additional **pay** actions while the turnstile remains open.
- The entire sequence can repeat any number of times.

Example Accepted Words

p	(Pay once, unlocks, and stays open)
pp	(Pay twice, remains unlocked)
pup	(Pay, push to lock, then pay again to unlock)
$ppuppp$	(Multiple pays, a push to lock, then more pays)
$upupup$	(Invalid pushes in locked state, followed by pays to unlock)

Exercise: Word Classification in Languages Determine for the following words if they are contained in L_1, L_2, or L_3.

Here, based on the descriptions given later in the report:

- L_1 is the set of words that contain the substring "01".
- L_2 is the set of words whose lengths are powers of two.
- L_3 is the set of words with an equal number of 0s and 1s.

The words under consideration are:

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

Explanation:

- $w_1 = 10011$: Contains the substring "01" (specifically, the third and fourth symbols form "01"). Its length is 5 (not a power of two), and it has 3 ones versus 2 zeros.
- $w_2 = 100$: Does not contain the substring "01" (the pairs are "10" and "00"), its length is 3 (not a power of two), and it has 1 one and 2 zeros.
- $w_3 = 10100100$: Contains "01" (for example, the second and third symbols form "01"); its length is 8 (which is 2^3); however, it has 3 ones and 5 zeros.
- $w_4 = 1010011100$: Contains "01" (e.g., the first occurrence between the first and second symbols or elsewhere); its length is 10 (not a power of two); and it has 5 ones and 5 zeros.
- $w_5 = 11110000$: Does not contain the substring "01" (the transition from 1s to 0s gives "10" rather than "01"); its length is 8 (a power of two); and it has 4 ones and 4 zeros.

Exercise: DFA Run Acceptance Consider the DFA from above (see dfa_example). Consider the paths corresponding to the words $w_1 = 0010$, $w_2 = 1101$, and $w_3 = 1100$. For which of these words does their run end in the accepting state?

Definition. We call

$$L(\mathcal{A}) := \{w \in \Sigma^* \mid \text{The run for } w \text{ in } \mathcal{A} \text{ ends in some } q \in F\}$$

the language *accepted* by $\mathcal{A}$.

Answer: After tracing the transitions in the given DFA, we find that:

- For $w_1 = 0010$, the run ends in a non-accepting state.
- For $w_2 = 1101$, the run ends in a non-accepting state.
- For $w_3 = 1100$, the run ends in an accepting state.

One plausible interpretation (consistent with common examples such as a DFA for determining divisibility by 3 or for ensuring an even number of 1s) shows that only w_3 meets the acceptance condition, while w_1 and w_2 do not.

Summary of Week 1

Finite automata are abstract machines used to recognize regular languages, which can be fully described using finite-state transitions. This chapter explores deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs), demonstrating that NFAs, despite their flexibility, recognize the same class of languages as DFAs.

A key distinction between the two is that DFAs have a single active state at any time, whereas NFAs may simultaneously exist in multiple states. While NFAs simplify language representation, they can always be converted into equivalent DFAs through an algorithmic transformation.

An important extension of NFAs allows transitions on the empty string, further enhancing their expressiveness while still recognizing only regular languages. These ε-NFAs will later play a crucial role in proving the equivalence of finite automata and regular expressions.

The chapter also introduces an applied perspective on finite automata through a real-world example: electronic money protocols. By modeling interactions between a bank, a store, and a customer as finite automata, the protocol's correctness and potential fraud vulnerabilities can be analyzed. The study concludes with constructing a product automaton to validate interactions, ensuring that transactions occur as intended while preventing unauthorized duplication or cancellation of funds.

2.1.4 Conclusion

Deterministic Finite Automata (DFAs) are a fundamental concept in automata theory, providing a mathematical model for recognizing patterns in strings. A DFA is defined as a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an input alphabet, δ is the transition function mapping states and symbols to new states, q_0 is the initial state, and F is the set of accepting states. DFAs process input strings deterministically, meaning that for every state and input symbol, there is exactly one transition.

Formal languages are sets of words over an alphabet Σ. The set of all possible words is denoted Σ^*, which includes the empty word ε. The length of a word w is written as $|w|$, and the occurrence of a symbol a in w is denoted $|w|_a$. Example languages include L_1, which contains words with the substring "01," L_2, which consists of words whose lengths are powers of two, and L_3, which contains words with an equal number of 0s and 1s.

DFAs determine if a word belongs to a language by processing transitions. If the final state is in F, the word is accepted; otherwise, it is rejected.

2.2 Week 2: DFA Implementation in Python

2.2.1 Introduction

In this homework, we explore the implementation of deterministic finite automata (DFAs) in Python. We will go beyond the simple graphical representation of DFAs and build them using Python types.

2.2.2 Readings

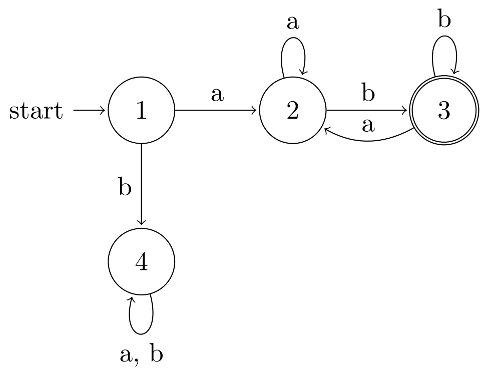
Chapter 2.2.4 A **Deterministic Finite Automaton (DFA)** is a formal model of computation that processes input sequences while maintaining a single, well-defined state at any given time. The term "deterministic" means that for each input symbol, the automaton transitions to exactly one state.

2.2.3 Homework

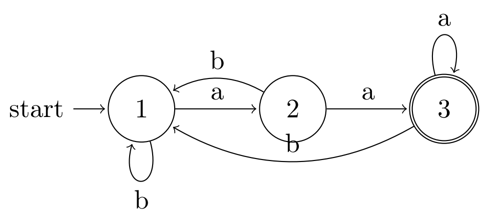
Implementing DFAs in Python We will implement the following DFAs in Python using this `DFA` class:

```
class DFA:

    def __init__(self, Q, Sigma, delta, q0, F):
        self.Q = Q  # Set of states
        self.Sigma = Sigma  # Alphabet
        self.delta = delta  # Transition function (dict)
        self.q0 = q0  # Initial state
        self.F = F  # Set of accepting states

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

    def run(self, w):
        current_state = self.q0  # Start at initial state
        loop = 0
        for symbol in w:
            loop += 1
            if symbol not in self.Sigma:
                return False  # Reject if symbol is not in the alphabet
            if (current_state, symbol) not in self.delta:
                return False  # Reject if there’s no valid transition
            current_state = self.delta[(current_state, symbol)]  # Move to next state

        return current_state in self.F  # Accept if final state is in F
```

Exercise 1: Word Processing with DFAs



Automaton $\mathcal{A}_1$

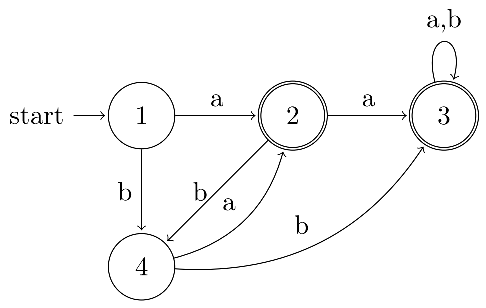


Automaton $\mathcal{A}_2$

Here is how we initialize the DFAs in Python:

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

# DFA A2
Q = {1,2,3}
Sigma = {’a’,’b’}
delta = {(1,’a’):2, (1,’b’):1, (2,’a’):3, (2,’b’):1,
         (3,’a’):3, (3,’b’):1}
q0 = 1
F = {3}
A2 = dfa.DFA(Q, Sigma, delta, q0, F)
```

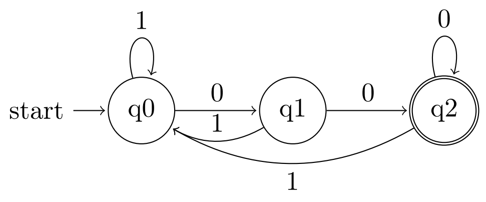
Constructing the Complement DFA To construct an automaton A_0 that accepts exactly the words that A refuses and vice versa we can simply swap the accepting states with the previously non-accepting states.



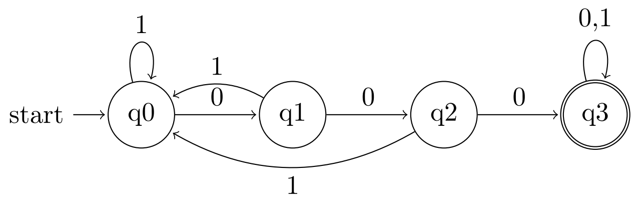
We can represent this operation with a `refuse()` function in Python:

```
def refuse(A):
    """Constructs a DFA A0 that accepts exactly the words that A refuses and vice versa."""
    Q0 = A.Q
    Sigma0 = A.Sigma
    delta0 = A.delta
    q0_0 = A.q0
    F0 = Q0 - A.F  # Complement of the accepting states

    return dfa.DFA(Q0, Sigma0, delta0, q0_0, F0)
```

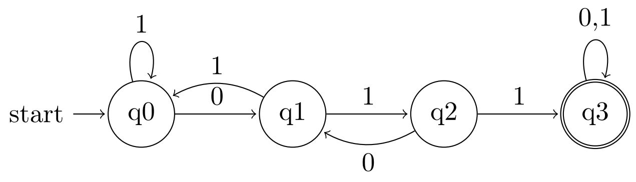
Exercise 2.4.4 (a) DFA for strings ending in 00



(b) DFA for strings containing 000 as a substring



(c) DFA for strings containing 011 as a substring



2.2.4 Conclusion

A **Deterministic Finite Automaton (DFA)** is a theoretical computational model used to recognize formal languages. A DFA consists of a finite set of states Q, an input alphabet Σ, a transition function $\delta : Q \times \Sigma \to Q$, a start state q_0, and a set of accepting states F. The machine processes strings sequentially, transitioning between states according to δ. If, after consuming the entire string, the DFA ends in an accepting state, the input is accepted; otherwise, it is rejected.

2.3 Week 3: DFAs and NFAs

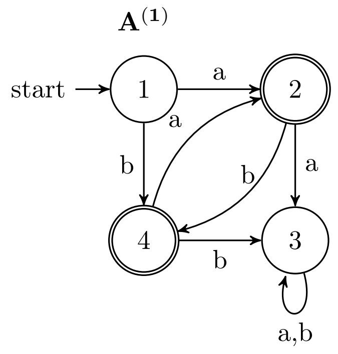
2.3.1 Introduction

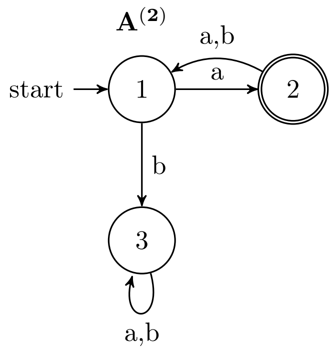
This week we covered ways to combine automata. We focused on the set theory used in the combination as well as some notation. Lastly, we introduced Nondeterministic Finite Automatas, or NFAs.

2.3.2 Homework

Extended Transition Functions **Brief summary of the question:** Given the two DFAs below:

- Compute the extended transition functions $\hat{\delta}^{(1)}(1,\, abaa)$ and $\hat{\delta}^{(2)}(1,\, abba)$, showing all steps.
- Describe the language accepted by each automaton.





$$L\big(A^{(1)}\big) = \{\, w \in \{a, b\}^{+} \mid \text{no two consecutive symbols are the same}\},$$

$$L\big(A^{(2)}\big) = a\,((a \mid b)\,a)^{*} = \{\, w \in \{a, b\}^{*} \mid w \text{ has odd length, starts with } a, \text{ and every odd position is } a\}.$$

$$\hat{\delta}^{(1)}(1,\, abaa) = 3, \quad \hat{\delta}^{(2)}(1,\, abba) = 3.$$

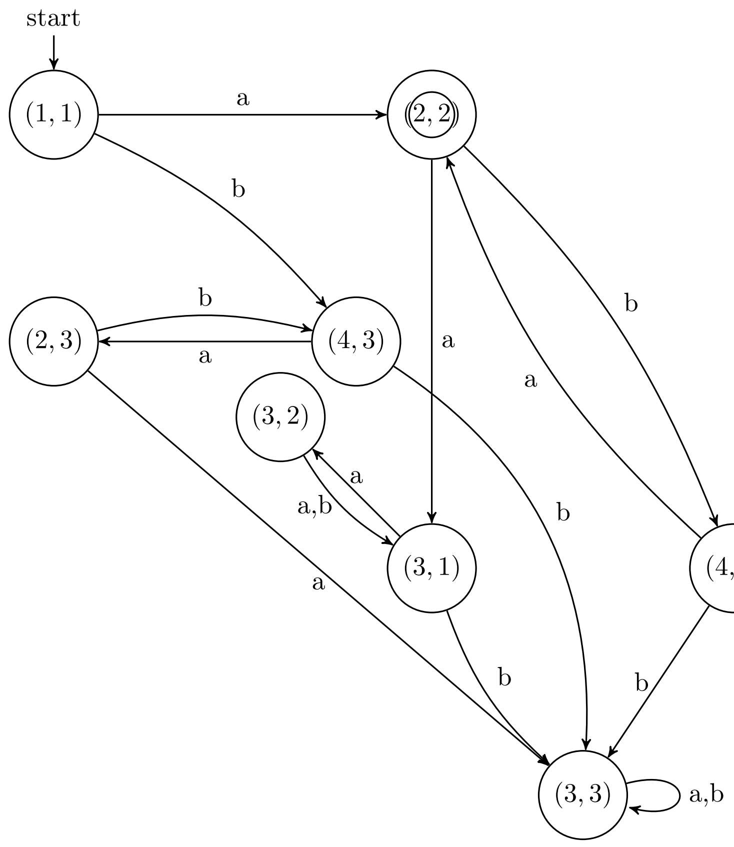
Intersubsection Automaton for $A^{(1)}$ and $A^{(2)}$ We form the product automaton

$$A = (Q^{(1)} \times Q^{(2)},\, \Sigma,\, \delta,\, (q_0^{(1)}, q_0^{(2)}),\, F^{(1)} \times F^{(2)}),$$

where

$$Q^{(1)} = \{1,2,3,4\},\ Q^{(2)} = \{1,2,3\},\ \Sigma = \{a,b\},$$

$\delta((p,q),x) = (\delta^{(1)}(p,x), \delta^{(2)}(q,x))$, and accepting states $F^{(1)} \times F^{(2)}$.



Why $L(A) = L(A^{(1)}) \cap L(A^{(2)})$? A simulates both in parallel and accepts iff both coordinates are accepting.

Changing to Obtain Union Use the same transitions but set

$$F' = (F^{(1)} \times Q^{(2)}) \,\cup\, (Q^{(1)} \times F^{(2)}),$$

so $L(A') = L(A^{(1)}) \cup L(A^{(2)})$.

Exercise 2.2.7

Claim. If $\delta(q,a) = q$ for all a, then $\delta(q,w) = q$ for any w.

Proof (by induction): Base: $\delta(q,\varepsilon) = q$. Step: $\delta(q,xa) = \delta(\delta(q,x),a) = \delta(q,a) = q$.

2.3.3 Readings

Chapter 2.3 The extended transition function for an NFA,

$$\hat{\delta}(q,\varepsilon) = \{q\}, \quad \hat{\delta}(q,\, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p,a),$$

captures nondeterminism. The subset construction converts an NFA N into a DFA D whose states are subsets of N's states, preserving $L(D) = L(N)$.

2.3.4 Conclusion

Throughout these problems and readings, we deepened our understanding of DFAs and NFAs: extended transition functions, intersubsection/union constructions, and the subset construction proving NFA–DFA equivalence.

Interesting question: Maximal Blow-Up in the Subset Construction.
Describe an n-state NFA forcing all 2^n subsets to appear in its equivalent DFA and prove minimality.

2.4 Week 4: Determinization

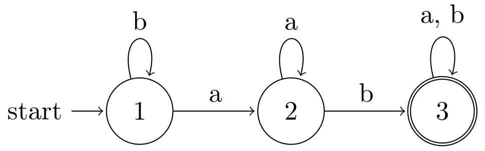
2.4.1 Introduction

In this report, we explore fundamental aspects of both deterministic and nondeterministic finite automata (DFAs and NFAs), including extended transition functions, product automaton construction for intersubsection, and state modifications for union. We further demonstrate the subset construction to establish the equivalence of NFAs and DFAs, showcasing the broad utility of these automata concepts in both theoretical and practical contexts.

2.4.2 Homework

Homework 1 Let $\mathcal{A} = (Q, \Sigma, \delta : Q \times \Sigma \to Q, q_0, F)$ be a DFA. Explain in what way you can view $\mathcal{A}$ as an NFA by doing the following:

1. Let $\mathcal{A}$ denote the following DFA:



   Here:

$$\begin{aligned} Q = \{1, 2, 3\}, &\quad \Sigma = \{a, b\}, \\ q_0 = 1, &\quad F = \{3\}, \\ \delta(1, a) = 2, &\ \ \delta(1, b) = 1, \\ \delta(2, a) = 2, &\ \ \delta(2, b) = 3, \\ \delta(3, a) = 3, &\ \ \delta(3, b) = 3. \end{aligned}$$

   Explain how you can understand $\mathcal{A}$ also as an NFA.

2. More generally, let $\mathcal{A} = (Q, \Sigma, \delta : Q \times \Sigma \to Q, q_0, F)$ be a DFA. Define an NFA

$$\mathcal{A}' = (Q', \Sigma, \delta' : Q' \times \Sigma \to \mathcal{P}(Q'), q_0', F')$$

   such that $L(\mathcal{A}) = L(\mathcal{A}')$.

3. Justify why your construction satisfies the desired condition.

Solutions

1) Viewing the Example DFA as an NFA A deterministic finite automaton (DFA) can be seen as a special case of a nondeterministic finite automaton (NFA) by interpreting its transition function in the following way: in an NFA, the transition function

$$\delta' : Q \times \Sigma \to \mathcal{P}(Q)$$

produces *sets* of possible next states. In a DFA, however, for each state q and input symbol a, there is exactly one next state $\delta(q, a)$. To view the DFA as an NFA, we simply set:

$$\delta'(q, a) = \{\delta(q, a)\}.$$

Hence, each transition in the original DFA becomes a transition to a *singleton set* in the NFA, preserving the recognized language because no extra nondeterminism is introduced.

2) General Construction from a DFA to an NFA Given any DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, define an NFA $\mathcal{A}' = (Q', \Sigma, \delta', q_0', F')$ by:

$$\begin{aligned} Q' &= Q, \\ q_0' &= q_0, \\ F' &= F, \\ \delta'(q, a) &= \{\delta(q, a)\} \quad \text{for all } q \in Q,\ a \in \Sigma. \end{aligned}$$

3) Why $L(\mathcal{A}) = L(\mathcal{A}')$? Because each step in the DFA corresponds to a unique singleton transition in the NFA:

- If a string w is accepted by the DFA, the identical path exists in the NFA.
- If a string w is rejected by the DFA, no path in the NFA leads to acceptance.

Thus the two automata recognize the same language.

Homework 2

1. Describe in words the language $L(\mathcal{A})$ accepted by the NFA $\mathcal{A}$ pictured below.
2. Specify the automaton $\mathcal{A}$ formally as $(Q, \Sigma, \delta, q_0, F)$.
3. Using the extended transition function $\hat{\delta}$, compute $\hat{\delta}(q_0, 10110)$ step by step.
4. Find *all* paths in $\mathcal{A}$ for $v = 1100$ and $w = 1010$; represent each set of paths.
5. Construct the determinization $\mathcal{A}^D$ (power-set automaton) of $\mathcal{A}$.
6. Verify $L(\mathcal{A}) = L(\mathcal{A}^D)$. Is there a smaller DFA for the same language?

Solutions Below is the NFA $\mathcal{A}$:

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[loop above, "0{,}1"] \arrow[r, bend left, "1"] & q_1 \arrow[r, bend left, "1"] & q_2 \arrow[l, bend left, "0"] \arrow[r, bend left, "0"] & q_3 \arrow[loop above, "0{,}1"]
\end{tikzcd}$$

1) Language Description $\mathcal{A}$ accepts exactly those binary strings that can nondeterministically traverse $q_0 \to q_1 \to q_2$ on two 1's and then take a 0 to q_3, after which any symbols are allowed.

2) Formal Specification

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F), \quad Q = \{q_0, q_1, q_2, q_3\},\ \Sigma = \{0, 1\},\ F = \{q_3\},$$

with

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

3) Extended Transition on 10110 $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$, $\hat{\delta}(q_0, 1) = \{q_0, q_1\}$, $\hat{\delta}(\{q_0, q_1\}, 0) = \{q_0\}$, $\hat{\delta}(\{q_0\}, 1) = \{q_0, q_1\}$, $\hat{\delta}(\{q_0, q_1\}, 1) = \{q_0, q_1, q_2\}$.

4) All Paths for 1100 and 1010 Branches arise at each nondeterministic choice; any accepting path must include $q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_3$.

5) Determinization (Power-Set)

$$Q^D = \mathcal{P}(Q),\ q_0^D = \{q_0\},\ F^D = \{S : S \cap \{q_3\} \neq \varnothing\},\ \delta^D(S, a) = \bigcup_{s \in S} \delta(s, a).$$

6) Verification & Minimization By construction $L(\mathcal{A}^D) = L(\mathcal{A})$. The resulting DFA can then be minimized via standard algorithms.

2.4.3 Conclusion

By viewing any DFA as an NFA with singleton transitions and applying the subset (powerset) construction, we have shown that NFAs and DFAs recognize exactly the same class of languages. This determinization process not only establishes the theoretical equivalence of nondeterministic and deterministic finite automata but also provides a concrete algorithm for converting an NFA into an equivalent DFA, reinforcing the practical utility of these foundational automata concepts.

2.5 Week 5: Equivalence and Minimization of Automata

2.5.1 Introduction

This report focuses on Equivalence and Minimization of Automata, covering the process of determining whether two different DFA representations define the same language and how to construct the smallest possible DFA that accepts a given regular language. The table-filling algorithm is introduced as a systematic way to check equivalence of states within a DFA by distinguishing them through input strings. This allows us to merge equivalent states, reducing the number of states while preserving language recognition. We also examine the uniqueness of the minimal DFA and its theoretical guarantees. Finally, the report discusses the complexity of these procedures and their implications for automata design.

2.5.2 Readings

Chapter 4.4 – Equivalence and Minimization of Automata Chapter 4.4 covers the equivalence and minimization of automata, focusing on the problem of determining whether two descriptions of regular languages define the same language. The chapter introduces the table-filling algorithm, a method to determine when two states of a DFA are equivalent by examining how they transition on different input strings. If two states lead to the same accepting or rejecting states for all possible inputs, they are considered equivalent and can be merged into a single state. If not, they are distinguishable. This process allows us to construct a minimal DFA, the smallest DFA that accepts a given language. The minimization process follows a structured approach:

1. Eliminate unreachable states.
2. Partition the remaining states into equivalence classes.
3. Construct a new DFA where each equivalence class represents a single state.

A key result is that the minimal DFA is unique up to renaming of states. The chapter also explores the complexity of minimization, showing that the table-filling algorithm runs in $O(n^2)$ time. Additionally, it presents an alternative perspective on DFA equivalence testing by constructing a combined DFA from two automata and checking equivalence of their start states. The final part discusses NFA minimization and its inherent difficulties compared to DFAs, highlighting limitations of direct state-equivalence methods for NFAs.

2.5.3 Homework

Note: The following exercises test understanding of state equivalence, DFA minimization, and nonregularity proofs.

Exercise 3.2.1 Here is the DFA's transition table:

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

(a) $R_{ij}^{(0)}$

$R_{ij}^{(0)}$	$j=1$	$j=2$	$j=3$
$i=1$	$\varepsilon \cup 1$	0	$\varnothing$
$i=2$	1	$\varnothing$	0
$i=3$	$\varnothing$	1	$\varepsilon \cup 0$

(b) $R_{ij}^{(1)}$

$R_{ij}^{(1)}$	$j=1$	$j=2$	$j=3$
$i=1$	1^*	1^*0	$\varnothing$
$i=2$	1^+	1^+0	0
$i=3$	$\varnothing$	1	$\varepsilon \cup 0$

(c) $R_{ij}^{(2)}$

$R_{ij}^{(2)}$	$j=1$	$j=2$	$j=3$
$i=1$	$1^* \cup 1^*0\,(0+1)^*\,1^+$	1^*0	$1^*0\,(0+1)^*\,0$
$i=2$	1^+	1^+0	0
$i=3$	$\varnothing$	1	$\varepsilon \cup 0$

(d) Regular expression

$$(1+01)^*\,00\,(0+10)^*.$$

Exercise 3.2.2

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

(a) $R_{ij}^{(0)}$

$R_{ij}^{(0)}$	$j=1$	$j=2$	$j=3$
$i=1$	ε	0	1
$i=2$	0	ε	1
$i=3$	1	0	ε

(b) $R_{ij}^{(1)}$

$R_{ij}^{(1)}$	$j=1$	$j=2$	$j=3$
$i=1$	ε	0	1
$i=2$	0	$\varepsilon \cup 00$	$1 \cup 01$
$i=3$	1	$0 \cup 10$	$\varepsilon \cup 11$

(c) $R_{13}^{(2)}$

$$R_{13}^{(2)} = 1 \;\cup\; 0\,(00)^*\,(1+01).$$

(d) Regular expression

$$1 \;+\; 0\,(00)^*\,(1+01).$$

Exercise 4.1.1 Prove the following languages are not regular using the pumping lemma:

(a) $\{0^n1^n \mid n \geq 1\}$ **Proof.** Let p be the pumping length. Consider $s = 0^p1^p$. Any decomposition $s = xyz$ with $|xy| \leq p$ forces $y = 0^k$ for some $k \geq 1$. Pumping down ($i = 0$) yields $0^{p-k}1^p$, which has fewer 0s than 1s, so $0^{p-k}1^p \notin L$. Contradiction.

(b) Balanced parentheses **Proof.** Let p be the pumping length and take

$$s = \underbrace{((\dots(}_{p}\underbrace{)\dots))}_{p}.$$

Then $|s| \geq p$, and any decomposition $s = xyz$ with $|xy| \leq p$ has y consisting only of "(". Pumping down removes some "(" while leaving all ")", yielding an unbalanced string. Contradiction.

(c) $\{0^n 1 0^n \mid n \geq 1\}$ **Proof.** Let $s = 0^p 1 0^p$. Any $s = xyz$ with $|xy| \leq p$ again gives $y = 0^k$. Pumping down produces $0^{p-k} 1 0^p$, which no longer has equal numbers of leading and trailing 0s. Contradiction.

(d) $\{0^n 1^m 2^n \mid n, m \geq 0\}$ **Proof.** Take $s = 0^p 1 2^p$. Decomposition $s = xyz$ with $|xy| \leq p$ yields $y = 0^k$. Pumping down yields fewer 0s than 2s ($0^{p-k} 1 2^p$), so not in the language. Contradiction.

(e) $\{0^n 1^m \mid n \leq m\}$ **Proof.** Let $s = 0^p 1^p$. Any initial segment y lies in the 0-block; pumping up ($i = 2$) gives $0^{p+k} 1^p$ with more 0s than 1s, violating $n \leq m$. Contradiction.

(f) $\{0^n 1^{2n} \mid n \geq 1\}$ **Proof.** Let $s = 0^p 1^{2p}$. Decompose $s = xyz$ with $|xy| \leq p$, so $y = 0^k$. Pumping down yields $0^{p-k} 1^{2p}$, where $2(p - k) \neq 2p$, so string is rejected. Contradiction.

Exercise 4.1.2 Prove the following are not regular by the pumping lemma:

(a) $\{0^n \mid$ **n is a perfect square**$\}$ **Proof.** Let p be the pumping length and $s = 0^{p^2}$. Any decomposition with $|xy| \leq p$ has $y = 0^k$, $1 \leq k \leq p$. Pumping up gives a length $p^2 + (i - 1)k$ strictly between p^2 and $(p + 1)^2$, not a square. Contradiction.

(b) $\{0^n \mid n$ **is a perfect cube**$\}$ **Proof.** Same idea: the gap between consecutive cubes exceeds the maximum pump size p.

(c) $\{0^n \mid n$ **is a power of 2**$\}$ **Proof.** Between 2^p and 2^{p+1} there is no other power of two, and pumping by at most p cannot reach 2^{p+1}.

(d) Strings of length a perfect square **Proof.** Same as (a) but phrased in terms of string length.

(e) $\{ww \mid w \in \{0, 1\}^*\}$ **Proof.** Let $s = ww$ where $|w| = p$. Decompose in the first p symbols; pumping alters only the first half, breaking equality of halves.

(f) $\{ww^R \mid w \in \{0, 1\}^*\}$ **Proof.** Let $s = ww^R$ with $|w| = p$. Any pump in the first block misaligns the mirror image.

(g) $\{w\bar{w} \mid w \in \{0, 1\}^*\}$ **Proof.** Similar: pumping in w changes its length, so it no longer matches $\bar{w}$.

(h) $\{w1^n \mid |w| = n\}$ **Proof.** Let $s = 0^p 1^p$. Pumping in the 0-block yields $|w| \neq n$, so no longer in the language.

2.5.4 Conclusion

In this report, we explored methods for testing equivalence of automata and minimizing DFAs using the table-filling algorithm. We demonstrated how equivalent states can be merged into a unique minimal DFA, ensuring that no smaller DFA exists for the same language. We also applied the pumping lemma to rigorously prove that a wide variety of languages are not regular. These concepts are fundamental in automata theory, with applications in compiler design, pattern recognition, and formal verification. **Question:** *Is there a polynomial-time algorithm for minimizing NFAs? If not, what makes NFA minimization so much harder than DFA minimization?*

2.6 Weeks 6 and 7: Turing Machines and Decidability

2.6.1 Introduction

In Weeks 6 and 7, we extend our study from finite automata to the Turing machine as a more powerful model of computation (subsection 8.2) and employ it to demonstrate key limits of algorithmic decidability: we motivate undecidability via "hello–world" detection (subsection 8.1), prove the diagonal language $L_d = \{w_i \mid w_i \notin L(M_i)\}$ is not even r.e., and show the universal language $L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\}$ is r.e. but undecidable—complemented by exercises on designing simple Turing machines and classifying halting-related languages and their closure properties.

2.6.2 Readings

In this unit we introduce the Turing machine model (subsection 8.2) and use it to prove fundamental limits of computation:

- subsection 8.1 motivates undecidability via "hello–world" detection.
- subsection 9.1 defines the diagonal language

  $$L_d = \{\, w_i \mid w_i \notin L(M_i)\},$$

  showing it is not even r.e.
- subsection 9.2 studies the universal language

  $$L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\},$$

  proving it is r.e. but not decidable.

2.6.3 Exercises

Exercise A: Constructing Turing Machines **Problem.** Over the alphabet $\{0, 1\}$ with blank symbol `_`, design three machines:

1. M_1 accepts $\{1\,0^n \mid n \geq 0\}$ and on input $1\,0^n$ rewrites it to $1\,0^{n+1}$ then halts in the accept state.
2. M_2 accepts $\{1\,0^n \mid n \geq 0\}$ and on input $1\,0^n$ erases all zeros, leaving only a single '1', then halts accept.
3. M_3 accepts every binary string and on any input flips each '0' to '1' and each '1' to '0', then halts accept.

Paste each of the following into the simulator (use verbatim mode), compile, and run:

Machine `M1_Increment`

```
name: M1_Increment
init: q0
accept: q_accept

q0,1
q1,1,>
q1,0
q1,0,>
q1,_
q2,0,<
q2,0
q2,0,<
q2,1
q_accept,1,-
```

Machine `M2_EraseZeros`

```
name: M2_EraseZeros
init: q0
accept: q_accept

q0,1
q1,1,>
q1,0
q1,_,<
q2,_,<
q2,1
q_accept,1,-
```

Machine `M3_SwapBits`

```
name: M3_SwapBits
init: q0
accept: q_accept

q0,0
q0,1,>
q0,1
q0,0,>
q0,_
q_accept,_,-
```

Exercise 1: Halting-Related Languages Classify each language as decidable, r.e. only, or co-r.e. only:

$$\begin{aligned} L_1 &= \{ M \mid M \text{ halts on its own encoding}\}, \\ L_2 &= \{(M, w) \mid M \text{ halts on input } w\}, \\ L_3 &= \{(M, w, k) \mid M \text{ halts on } w \text{ within } k \text{ steps}\}. \end{aligned}$$

Answer.

- L_1: r.e. but not co-r.e. (undecidable).
- L_2: r.e. but not co-r.e. (undecidable).
- L_3: decidable (simulate up to k steps), hence both r.e. and co-r.e.

Exercise 2: Closure Properties Decide which of the following closure statements hold in general and justify your answer.

1. *If L_1, L_2 are decidable then $L_1 \cup L_2$ is decidable.*

   True. Let M_1, M_2 be Turing machines (TMs) that decide L_1 and L_2. On input w, run M_1; if it accepts, accept. Otherwise run M_2 and output its answer. Because both M_1 and M_2 always halt, this composite machine always halts and decides the union.

2. *If L is decidable then $\overline{L}$ is decidable.*

   True. If M decides L, construct M' that runs M and swaps its accept/reject outcomes. Since M halts on every input, so does M', hence $\overline{L}$ is decidable.

3. *If L is decidable then L^* is decidable.*

   True. Given a decider M for L, we can decide whether a string w belongs to L^* by nondeterministically cutting w into $k \geq 0$ blocks $w = w_1 \cdots w_k$ and checking each block with M. Deterministically we can do the same by trying all splits of w (there are $\leq |w| - 1$ of them) and running M on each piece—this brute-force procedure halts because M halts on every block and the search space is finite.

4. *If L_1, L_2 are r.e. then $L_1 \cup L_2$ is r.e.*

   True. Let M_1, M_2 semi-decide L_1, L_2. On input w run M_1 and M_2 in dovetailing fashion (e.g. alternate one step of each). If either machine accepts during the simulation, accept. If $w \in L_1 \cup L_2$ at least one of the two machines eventually accepts, so the dovetailing TM accepts; otherwise it runs forever, matching the r.e. definition.

5. *If L is r.e. then $\overline{L}$ is r.e.*

   False. The classic counter-example is the halting language

   $$L_H = \{\langle M, w \rangle \mid M \text{ halts on } w\},$$

   which is r.e. (simulate M on w and accept if it halts) but whose complement is not r.e.; otherwise both L_H and $\overline{L_H}$ would be r.e. and hence decidable, contradicting the undecidability of the halting problem.

6. *If L is r.e. then L^* is r.e.*

   True. Semi-decide L^* by dovetailing over all ways to split the input w into blocks $w_1 \cdots w_k$ *and* over all interleaved simulations of the r.e. machine M on each block. Accept if every block in one of the splits is accepted by M. Because $w \in L^*$ implies at least one split with all blocks in L, some branch of the dovetailing search eventually accepts.

2.6.4 Conclusion

We have provided complete Turing-machine configurations for the simulator, classified key halting-related languages, and reviewed closure properties of decidable and r.e. classes. These results illustrate the power and limits of Turing machines and set the stage for complexity-theoretic questions in subsequent weeks.

2.7 Week 8 & 9: Complexity Theory, Growth Comparisons, and Sorting Runtimes

2.7.1 Introduction

We begin with an overview of intractable computational problems from Chapter 10 of Hopcroft, Motwani, and Ullman, introducing the complexity classes P and NP and the notion of NP-completeness. Then we work through exercises on ordering and relating functions by asymptotic growth and finish with classic sorting-algorithm comparisons. Along the way we include a couple of illustrative plots to make the abstract inclusions concrete.

2.7.2 Readings

Chapter 10.1: The Classes P and NP Chapter 10.1 refocuses from mere decidability to *tractability*: among the decidable problems, which can actually be solved in a reasonable amount of time? It formalizes the class P as those languages decidable by a deterministic Turing machine in time polynomial in the input length, and NP as those decidable by a nondeterministic machine whose every branch halts in polynomial time. The chapter motivates polynomial time as the practical threshold—algorithms superpolynomial in nature blow up too quickly to handle large inputs—and introduces polynomial-time reductions, which transform instances of one problem into another in polynomial time, preserving membership. These reductions provide a rigorous way to compare problem hardness: if A reduces to B and B is in P, then A is also in P. Finally, the chapter highlights problems whose best-known solutions are exponential, hinting at the P versus NP question at the heart of complexity theory.

Chapter 10.2: SAT and Cook's Theorem Chapter 10.2 presents the Boolean Satisfiability Problem (SAT): given a propositional formula built from variables, negation, conjunction, and disjunction, does there exist an assignment of true/false values that makes it true? After fixing a finite-alphabet encoding, it shows SAT lies in NP by having a nondeterministic machine guess an assignment and then evaluate the formula in polynomial time. The centerpiece is Cook's Theorem, which constructs, in polynomial time, a Boolean formula whose structure enforces that an NP machine's accepting computation on input x exists if and only if the formula is satisfiable. By encoding the entire computation tableau into variables and clauses that ensure correct transitions and an accepting state, the proof establishes that every NP problem reduces to SAT. Thus SAT is NP-complete: it sits in NP and is as hard as any NP problem.

Chapter 10.3: CNF, CSAT, and 3SAT Chapter 10.3 refines SAT by restricting formula shape to *conjunctive normal form* (CNF)—an AND of clauses, each a disjunction of literals—and to k-CNF where each clause has exactly k literals. Converting arbitrary formulas to equivalent CNF can blow up exponentially, so the chapter gives an *equisatisfiable* transformation: push negations downward via De Morgan's laws so they only apply to variables, then introduce fresh variables to break complex subformulas into small clauses without replicating subexpressions. This yields a polynomial-time reduction from general SAT to CSAT (CNF-SAT), proving CSAT NP-complete. Finally, it shows how to transform any CNF into an equisatisfiable 3-CNF formula in linear time by splitting long clauses with auxiliary variables. Hence even 3SAT remains NP-complete, cementing its role as the canonical hard problem in complexity theory.

2.7.3 Homework

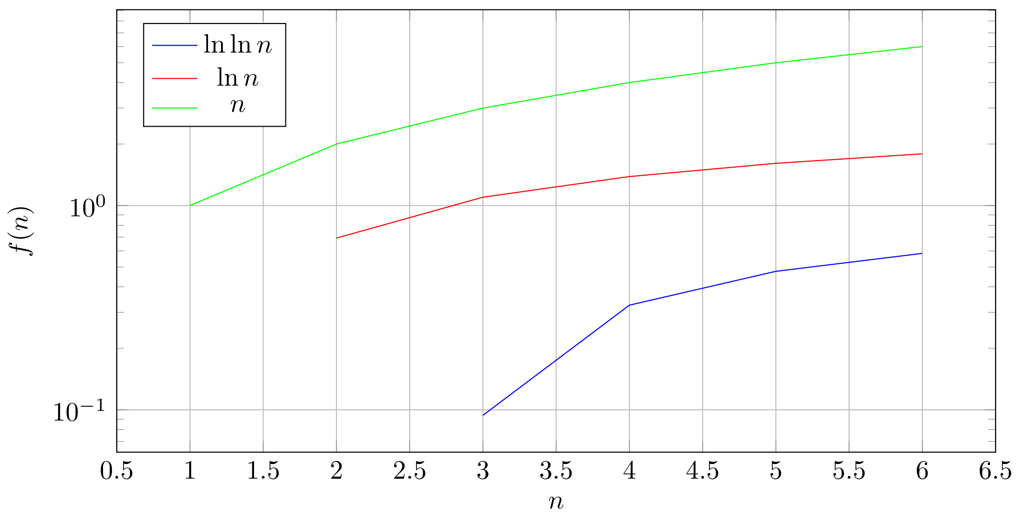
Exercise 1 Order by growth (slow $\to$ fast):

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

Answer.

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

Growth Plot



Exercise 2 For $f, g, h\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, prove:

1. $f \in O(f)$,
2. $O(c\,f) = O(f) \quad (\forall c > 0)$,
3. $f(n) \leq g(n)$ eventually $\implies O(f) \subseteq O(g)$,
4. $O(f) \subseteq O(g) \implies O(f+h) \subseteq O(g+h)$,
5. $h(n) > 0\ \forall n,\ O(f) \subseteq O(g) \implies O(f\,h) \subseteq O(g\,h)$.

Proof.

1. $f(n) \leq 1 \cdot f(n)$, so $f \in O(f)$.
2. If $u \in O(c\,f)$, then $u(n) \leq C\,c\,f(n)$, hence $u \in O(f)$. Conversely similarly.
3. If $f(n) \leq g(n)$ for large n and $u \in O(f)$, then $u(n) \leq C\,f(n) \leq C\,g(n)$, so $u \in O(g)$.
4. If $u \in O(f)$, then $u(n) + h(n) \leq C\,f(n) + h(n) \leq C\big(f(n) + h(n)\big)$, giving $u + h \in O(f+h) \subseteq O(g+h)$.
5. If $u \in O(f)$ and $h(n) > 0$, then $u(n)\,h(n) \leq C\,f(n)\,h(n)$, so $u\,h \in O(f\,h) \subseteq O(g\,h)$.

Exercise 3 Let $i, j, k, n \in \mathbb{N}$. Prove:

1. $j \leq k \implies O(n^j) \subseteq O(n^k)$,
2. $j \leq k \implies O(n^j + n^k) \subseteq O(n^k)$,
3. $O\Big(\sum_{m=0}^{k} a_m\, n^m\Big) = O(n^k)$,
4. $O(\log n) \subseteq O(n)$,
5. $O(n \log n) \subseteq O(n^2)$.

Proof.

1. $n^j \leq n^k$ for $n \geq 1$, so $O(n^j) \subseteq O(n^k)$.
2. For $n \geq 1$, $n^j + n^k \leq 2n^k$, giving $O(n^j + n^k) \subseteq O(n^k)$.
3. $\sum_{m=0}^{k} a_m n^m \leq \left(\sum |a_m|\right) n^k$.
4. For $n \geq 2$, $\log n \leq n$.
5. For $n \geq 2$, $n \log n \leq n^2$.

Exercise 4 Which relationships hold between:

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

Answer.

1. $O(\sqrt{n}) \subsetneq O(n)$.
2. $O(n^2) \subsetneq O(2^n)$.
3. $O(\log n) \subseteq O((\log n)^2)$.
4. $O(2^n) \subsetneq O(3^n)$.
5. $O(\log_2 n) = O(\log_3 n)$.

Exercise 5 Classic sorting comparisons:

bubble sort vs. insertion sort, insertion sort vs. merge sort, merge sort vs. quick sort.

Discussion.

- **Bubble vs. Insertion:** Both worst-case $O(n^2)$, but insertion sort is $O(n)$ on nearly-sorted input.
- **Insertion vs. Merge:** Insertion sort worst-case $O(n^2)$, merge sort always $O(n \log n)$.
- **Merge vs. Quick:** Merge sort is $O(n \log n)$ always; quick sort is $O(n \log n)$ average, $O(n^2)$ worst, but often faster in practice.

2.7.4 Conclusion

We've surveyed P vs. NP, NP-completeness via SAT, practiced ordering and relating functions by asymptotic growth, and applied these insights to sorting algorithms. The included plots illustrate constant-factor and growth-rate comparisons without overwhelming the presentation.

Question: What's the fastest known quantum sorting algorithm in the query (comparison) model, and how close is it to being practical?

2.8 Weeks 10 & 11: Intractable Problems

2.8.1 Introduction

In Weeks 10 and 11 we shift our focus to computational intractability by defining the classes P and NP and the concept of polynomial-time reductions (Chapter 10.1), proving the NP-completeness of SAT via Cook's Theorem (Chapter 10.2), and then applying these ideas through exercises in rewriting formulas into CNF, deciding small SAT instances, and encoding complex constraints like Sudoku into conjunctive normal form.

2.8.2 Readings

Chapter 10.1: The Classes P **and** NP We refine decidability to *tractability.* A language L is in P if some deterministic TM decides it in time $O(n^k)$. A language L is in NP if some nondeterministic TM accepts it in time $O(n^k)$. We introduce *polynomial–time reductions*— transformations computable in polynomial time—and define *NP-completeness*:

Definition 2.8.1. A language L is *NP-complete* if

1. $L \in \mathrm{NP}$, and
2. for every $L' \in \mathrm{NP}$, there is a polynomial-time reduction $L' \leq_p L$.

Chapter 10.2: SAT and NP-Completeness

The SAT Problem. Boolean formulas are built from variables, $\wedge, \vee, \neg$, and parentheses. A formula is *satisfiable* if some assignment of $\{\mathrm{TRUE}, \mathrm{FALSE}\}$ makes it true. The language

$$\mathrm{SAT} \;=\; \big\{\varphi \mid \varphi \text{ a satisfiable Boolean formula}\big\}$$

is in NP (guess an assignment, evaluate in polynomial time).

Cook's Theorem. Every $L \in \mathrm{NP}$ reduces to SAT in polynomial time; hence SAT is NP-complete.

2.8.3 Homework

Exercise 1: Rewriting in CNF Rewrite each formula into an equivalent conjunctive normal form.

$$\varphi_1 := \neg\big((a \wedge b) \;\vee\; (\neg c \wedge d)\big), \quad \varphi_2 := \neg\big((p \vee q) \to (r \wedge \neg s)\big).$$

Solution.

$$\begin{aligned} \varphi_1 &= \neg\big(X \vee Y\big) && \text{where } X = (a \wedge b),\ Y = (\neg c \wedge d) \\ &= \neg X \;\wedge\; \neg Y \\ &= (\neg a \;\vee\; \neg b) \;\wedge\; (c \;\vee\; \neg d). \end{aligned}$$

$$\begin{aligned} \varphi_2 &= \neg\Big(\neg(p \vee q) \;\vee\; (r \wedge \neg s)\Big) && \big(P \to Q \equiv \neg P \vee Q\big) \\ &= \neg\big(\neg X \;\vee\; Y\big) && \text{where } X = (p \vee q),\ Y = (r \wedge \neg s) \\ &= X \;\wedge\; \neg Y \\ &= (p \vee q) \;\wedge\; (\neg r \vee s). \end{aligned}$$

Exercise 2: Satisfiability For each formula, decide whether it is satisfiable. If yes, give a satisfying assignment; if not, prove no such assignment exists.

$$\begin{aligned} \psi_1 &:= (a \vee \neg b) \;\wedge\; (\neg a \vee b) \;\wedge\; (\neg a \vee \neg b), \\ \psi_2 &:= (\neg p \vee q) \;\wedge\; (\neg q \vee r) \;\wedge\; \neg(\neg p \vee r), \\ \psi_3 &:= (x \vee y) \;\wedge\; (\neg x \vee y) \;\wedge\; (x \vee \neg y) \;\wedge\; (\neg x \vee \neg y). \end{aligned}$$

Solution.

- ψ_1: take $a = \mathrm{FALSE}$, $b = \mathrm{FALSE}$. Then

$$(0 \vee 1) \wedge (1 \vee 0) \wedge (1 \vee 1) = 1 \wedge 1 \wedge 1 = 1.$$

  Hence ψ_1 is satisfiable.

- ψ_2: note
$$\neg(\neg p \vee r) \equiv (p \wedge \neg r).$$
So $\psi_2 \equiv (\neg p \vee q) \wedge (\neg q \vee r) \wedge p \wedge \neg r$. From p and $\neg r$ we get q by the first clause, but then $\neg q \vee r$ becomes 0, contradiction. Thus ψ_2 is unsatisfiable.
- ψ_3: one checks all four assignments to (x, y) and finds each violates some clause. Hence ψ_3 is unsatisfiable.

Exercise 3: Sudoku CNF Encoding We have boolean variables
$$x_{r,c,v}, \quad r, c, v \in \{1, \dots, 9\},$$
true exactly when cell (r, c) holds value v. The CNF is $\bigwedge_{k=1}^{6} C_k$, where:

$$\begin{array}{lll}
C_1: & \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \big(x_{r,c,1} \vee x_{r,c,2} \vee \cdots \vee x_{r,c,9}\big), & \text{(each cell has at least one value)} \\
C_2: & \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \bigwedge_{\substack{v,w=1 \\ v \neq w}}^{9} \neg\big(x_{r,c,v} \wedge x_{r,c,w}\big), & \text{(each cell has at most one value)} \\
C_3: & \bigwedge_{r=1}^{9} \bigwedge_{v=1}^{9} \big(x_{r,1,v} \vee \cdots \vee x_{r,9,v}\big), & \text{(each row contains each value)} \\
C_4: & \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{9} \big(x_{1,c,v} \vee \cdots \vee x_{9,c,v}\big), & \text{(each column contains each value)} \\
C_5: & \bigwedge_{b_r=0}^{2} \bigwedge_{b_c=0}^{2} \bigwedge_{v=1}^{9} \big(x_{3b_r+1,\,3b_c+1,\,v} \ \vee \ \cdots \ \vee \ x_{3b_r+3,\,3b_c+3,\,v}\big), & \text{(each } 3 \times 3 \text{ block has each value)} \\
C_6: & \bigwedge_{(r,c) \in \text{Clues}} x_{r,c,v_{r,c}}, & \text{(respect the given clues).}
\end{array}$$

2.8.4 Conclusion

Weeks 8 & 9 introduced the classes P and NP, the notion of polynomial–time reduction, and NP-completeness. Cook's Theorem places SAT at the heart of intractability. We practiced CNF rewriting, satisfiability proofs, and large-scale CNF encodings (Sudoku), reinforcing both the theoretical limits and the practical modeling power of SAT. **Interesting Question** Is there a practical sub-exponential algorithm for SAT on random formulas, and how does it relate to the worst-case NP-completeness barrier?

2.9 Weeks 12 & 13: Graph Theory

2.9.1 Introduction

During these two weeks we pivot from abstract graph notions to their most celebrated applications in decision-science. Chapter 14 of Deo is our guide: it begins with transport networks and the classic *max-flow / min-cut* duality, then systematically injects the complications that arise in practice—multiple sources and sinks, vertex capacities, bidirectional (undirected) lines, mandatory lower-bound throughputs, and lossy transmission. The chapter culminates in two optimisation mainstays: obtaining the cheapest way to ship a required amount (the transportation problem) and orchestrating several commodities that must share the same network (multicommodity flow). At each stage Deo emphasises fast combinatorial algorithms and explains why they can outperform general linear-programming approaches. [Deo]

2.9.2 Readings

14.1 Transport Networks subsection 14.1 introduces *transport networks*—weighted directed graphs whose edge capacities model the maximum rate at which a commodity can move between facilities. A **flow** assigns non-negative amounts to edges while conserving quantity at every intermediate vertex and achieving value w from a unique source s to sink t. Deo formalises the *maximal-flow* problem, sketches the linear-programming formulation, and then proves the **Max-Flow Min-Cut Theorem**: the greatest achievable flow equals the minimum capacity of any $s-t$ cut. A constructive proof underlies the classic augmenting-path algorithm that iteratively raises the flow until no augmenting path remains. [Deo]

14.2 Extensions of Max-Flow Min-Cut subsection 14.2 generalises the basic theorem to more complex networks.

- *Multiple sources/sinks*: a supersource and supersink convert the instance to the single-commodity case, unless each source must reach a particular sink (the multicommodity variant).
- *Vertex capacities*: split a capacitated vertex into an in-vertex and out-vertex linked by an edge whose capacity equals the vertex limit.
- *Undirected edges*: replace each undirected edge by two opposite directed edges that share the same capacity.
- *Lower bounds*: allow each edge to require at least b_{ij} units of flow and derive feasibility/optimality conditions in terms of modified cuts.
- *Lossy networks*: assign an efficiency factor λ_{ij} so the out-flow equals λ_{ij} times the in-flow; analogous optimal-flow results hold.

These adaptations preserve the cut-flow duality or indicate when additional constraints break it. [Deo]

14.3 Minimal-Cost Flows subsection 14.3 attaches a per-unit cost d_{ij} to every edge and asks for the cheapest flow of prescribed value w. Deo connects this *transportation problem* with linear programming but emphasises a combinatorial solution: iteratively send flow along the least-cost *unsaturated* $s-t$ path, where path cost counts forward-edge expenses minus backward-edge savings. A key optimality lemma shows that augmenting by the cheapest unsaturated path always preserves minimality; the process terminates when no such path exists, yielding a minimal-cost maximal flow. [Deo]

14.4 Multicommodity Flow subsection 14.4 tackles networks that simultaneously carry k distinct commodities, each with its own source and sink but sharing edge capacities. Edge constraints become $\sum_{m=1}^{k} f_{ij}^{(m)} \leq c_{ij}$, and flow conservation holds for every commodity. The chapter highlights two fundamental tasks: (i) maximise the *total* shipped amount $\sum_m w_m$ and (ii) decide feasibility for specified $(w_1, \ldots, w_k)$. Unlike the single-commodity case, no simple cut characterisation exists in general—only special settings (e.g. two commodities in an undirected graph) admit a max-flow min-cut analogue. The discussion underscores the added complexity that competition for capacity introduces. [Deo]

2.9.3 Exercises

Exercise 1 (Max-flow & minimal-cut)

Network description. The directed graph has source s and sink t with capacities shown as "flow / capacity":

$$\begin{tikzcd}
s \arrow[r, "8/10"] \arrow[d, "0/10"'] & a \arrow[r, "0/4"] \arrow[dl, "0/2"] \arrow[d, "8/8"'] & c \arrow[d, "0/10"] \\
b \arrow[r, "0/9"] & d \arrow[r, "8/10"] \arrow[ur, "0/6"] & t
\end{tikzcd}$$

The initial flow has value 8.

(1) Ford–Fulkerson augmentation.

1. Residual path $s \to b \to d \to t$ min residual $= 2 \Rightarrow$ add 2.
2. Residual path $s \to b \to d \to c \to t$ min residual $= 6 \Rightarrow$ add 6.
3. Residual path $s \to a \to c \to t$ min residual $= 2 \Rightarrow$ add 2.
4. Residual path $s \to b \to d \to a \to c \to t$ min residual $= 1 \Rightarrow$ add 1.

No further $s - t$ path exists in the residual graph, so the algorithm terminates with

$$\boxed{\text{maximum flow value } = 19}.$$

The resulting flow on each edge is

edge	flow/capacity	
$s \to a$	10/10,	$s \to b$ 9/10
$a \to b$	0/2,	$a \to c$ 3/4, $a \to d$ 7/8
$b \to d$	9/9,	$d \to c$ 6/6, $d \to t$ 10/10, $c \to t$ 9/10

(2) Minimal s–t cut. Vertices reachable from s in the final residual graph are $P = \{s, b\}$. Edges leaving P are $s \to a$ (10) and $b \to d$ (9); their total capacity is

$$c(P, \bar{P}) = 10 + 9 = 19,$$

which equals the maximum-flow value, so $(P, \bar{P})$ is a minimal cut.

(3) Uniqueness of a maximal flow. A maximum-flow *value* is unique, but the *flow pattern* need not be—different distributions on parallel augmenting paths can yield the same value. In this network, for example, one may shift up to one unit from the path $s \to a \to d \to t$ to $s \to a \to c \to t$ (while respecting capacities) and still obtain value 19.

Exercise 2 (An unknown algorithm)

```
fun unknown(n)
1. r := 0
2. for k := 1 to n−1 do
3.      for l := k+1 to n do
4.           for m := 1 to l do
5.                r := r + 1
6. return r
```

1) Returned value as a function of n. The innermost statement (line 5) executes once for every triple (k, l, m) that satisfies

$$1 \le k \le n-1, \qquad k+1 \le l \le n, \qquad 1 \le m \le l.$$

Hence

$$r(n) = \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \sum_{m=1}^{l} 1 = \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} l.$$

Fix l. It appears in the inner sum whenever $k < l$, i.e. for $k = 1, 2, \ldots, l-1$ (a total of $l-1$ choices). Therefore

$$r(n) = \sum_{l=2}^{n} l(l-1) = \sum_{l=1}^{n} (l^2 - l) = \Big(\sum_{l=1}^{n} l^2\Big) - \Big(\sum_{l=1}^{n} l\Big).$$

Using the given formulas $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ and $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, we obtain

$$r(n) = \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} = \frac{n(n+1)(2n-2)}{6} = \boxed{\frac{n(n+1)(n-1)}{3}}.$$

2) Worst-case running time (Big-O). The algorithm performs $\Theta\big(r(n)\big)$ basic operations. Since

$$r(n) = \frac{n(n+1)(n-1)}{3} = \frac{1}{3}n^3 + O(n^2),$$

the worst-case running time grows on the order of n^3:

$$\boxed{T_{\text{worst}}(n) = O(n^3).}$$

3 Synthesis

Looking back over the entire semester of CPSC-406, I see how the individual topics we covered—from finite automata and formal languages to undecidability, complexity theory, and finally, graph algorithms—have gradually formed a cohesive and comprehensive understanding of algorithm analysis. At first, the course started straightforwardly enough, reviewing concepts like deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs). Even though I'd seen some of these ideas before in previous coursework, writing them up formally forced me to pay closer attention to small details I'd previously overlooked. In particular, the precise definition of transitions, accepting states, and formal languages became clearer as I structured these ideas in LaTeX.

The transition from finite automata to Turing machines marked a significant step. Initially, Turing machines seemed unnecessarily complex compared to DFAs and NFAs. However, through exercises where we explicitly constructed and simulated these machines, it became clear why Turing machines are foundational in computer science. Their ability to capture all computable functions—while still being relatively simple—demonstrated clearly why they're the standard for formalizing the concept of computability. The Halting Problem was an especially critical moment for my understanding. At first, the idea of an undecidable problem felt abstract, but the clarity of the proof made a deep impression. I recognized how natural and fundamental this limitation was. Attempting to understand precisely why certain problems defy algorithmic solutions was challenging, yet it solidified my grasp of what "decidable" really means in practice.

The classes P, NP, and NP-complete initially felt abstract, but exploring concrete examples—particularly through Boolean satisfiability (SAT)—helped ground my understanding. I found Cook's theorem particularly insightful, especially the idea that SAT could encapsulate every other NP problem. The significance of polynomial-time reductions was somewhat vague until I actively applied them to real examples, such as encoding Sudoku puzzles into CNF. This exercise was both enjoyable and enlightening, highlighting the deep connections between theoretical complexity classes and real-world applications.

Moreover, comparing different complexity classes—like understanding why NP-complete problems are considered practically difficult—was fascinating. It emphasized the value of understanding the theoretical limits when working on real computational problems. Sorting algorithms, too, provided a valuable practical connection. Analyzing and comparing algorithmic runtimes through Big-O notation and asymptotic analysis showed me clearly how theory and practice intersect. Problems that seemed trivially solvable became significantly more interesting when viewed through the lens of efficiency and scalability.

Graph algorithms further solidified this link between theory and real-world utility. Concepts like maximum flow, minimum cut, and their variants appeared immediately practical and applicable. Working through these examples in detail, particularly with the Ford–Fulkerson algorithm, helped clarify concepts that had felt abstract earlier. It was satisfying to see how theoretical efficiency measures directly impact their practical feasibility in applications like network routing and optimization.

One of the most challenging aspects of the course was maintaining a sense of continuity between topics that initially appeared disconnected. However, as I synthesized these topics week by week, I realized how each concept naturally built on the previous one. The structured note-taking and reflection process provided essential scaffolding, enabling me to make connections that might otherwise have gone unnoticed. I was particularly struck by how seemingly unrelated theoretical concepts from automata theory to complexity theory, all fed into the broader goal of understanding algorithmic behavior and limitations.

Overall, the synthesis of my learning this semester was significantly shaped by actively engaging with the course material through regular writing, reflecting, and problem-solving. The combined effort of understanding theory, practicing problem-solving, and clearly communicating my ideas has not only strengthened my foundational knowledge but also prepared me to better approach complex algorithmic problems in future coursework, research, or industry work. This experience underscored the value of clearly articulating complex ideas, reinforcing the idea that deep understanding arises not only from reading and problem-solving but also from careful reflection and communication.

4 Evidence Of Participation

1. Quantum Uncertainty and Scientific Limits — Guest Lecture Reflection

I found the recent lecture at Chapman University very stimulating. Reflecting on the inherent limits of the scientific method generates a unique sense of existential contemplation. As someone who identifies philosophically with absurdist nihilism, confronting the boundary of scientific knowledge evokes both curiosity and an odd sense of existential dread. When I was younger, I firmly believed in a fully deterministic universe—a universe complex, yet entirely quantifiable. In this deterministic worldview, knowing the complete initial conditions of the universe would, in theory, allow one to predict with absolute certainty every future event. Under such assumptions, science appears concrete and fully capable of unveiling reality. However, quantum mechanics profoundly disrupts this clear-cut vision of reality.

Over the past year, my studies at Chapman in quantum mechanics have consistently challenged and reshaped my understanding. The concept of quantum uncertainty is particularly perplexing when applied to large, chaotic ensembles of particles. Examples of such systems include Saturn's moon Hyperion, Earth's oceans, or even individual humans. These systems, due to their chaotic nature, exhibit a profoundly uncertain and indeterminate future when considered through the lens of quantum mechanics. This probabilistic perspective aligns interestingly with my understanding of the de Broglie wavelength associated with every physical object.

According to quantum theory, each object possesses an associated wavelength, defined as Planck's constant divided by the object's momentum. Given that Planck's constant is extremely small—on the order of 10^{-34}—it becomes clear why macroscopic, fast-moving objects exhibit incredibly short wavelengths. Consequently, the quantum wave packets representing larger objects like humans appear sharply defined, with positions and momenta that seem essentially certain. Although strictly speaking every object, including humans, has a wavefunction, their macroscopic scale makes quantum wavelike behavior practically negligible. For instance, this negligible quantum behavior is why humans don't visibly diffract when passing through doorways. A simple calculation illustrates this clearly: a 70 kg human moving at 1 m/s through a 1 m-wide doorway yields a diffraction angle of roughly 10^{-35} radians. Even at a distance of one full light-year from the doorway, the separation between the first and second-order diffraction fringes would only be around $10^{-15}\,m$, significantly smaller than atomic scales. Hence, we observe no practical diffraction effects.

The implications of the de Broglie wavelength are also fundamental to Bohr's atomic model. In Bohr's model of the hydrogen atom, the valid electron radii and energy levels arise because electrons must form standing waves. Only radii corresponding to integer multiples of the electron's de Broglie wavelength allow for continuous wavefunctions; otherwise, the electron's wavefunction would exhibit discontinuities. Interestingly, attempting to apply this same logic to macroscopic systems like celestial orbits highlights important distinctions. For example, the Moon's de Broglie wavelength, given its enormous mass and velocity, is approximately $10^{-61}\,m$. This scale is far smaller even than the Planck length—the shortest meaningful length scale in physics—so quantum constraints like standing wave conditions effectively become meaningless, allowing the Moon to occupy orbits that appear continuous.

Despite this, the Moon still behaves as a quantum wave packet, albeit at an unobservable scale. This observation does not violate the Heisenberg uncertainty principle, since the Moon's momentum is so large that the product of uncertainties in position and momentum easily surpasses the fundamental quantum limit. Additionally, unlike Hyperion, the Moon's relatively stable orbit is less chaotic, meaning quantum uncertainties have minimal influence on its future position predictions. Less chaotic systems, in general, are less sensitive to small errors in initial conditions, resulting in more predictable trajectories.

Considering these concepts, it is fascinating (although likely flawed in certain assumptions) to contemplate the quantum effects on macroscopic celestial systems. In particular, the lecturer's mention of Hyperion intrigued me greatly. I am curious how quantum uncertainty was quantitatively incorporated into the calculations that predicted the breakdown of classical predictability within approximately 50 years. Understanding the exact methodology and assumptions involved would be enlightening and could further clarify the subtle interplay between quantum mechanics and chaotic macroscopic systems.

2. Video Summary – Grover's Algorithm and Quantum Speedup

In this video[1], the narrator critically examines common misconceptions surrounding quantum computing, particularly the oversimplified notion that quantum computers simply evaluate all possible bit sequences simultaneously via superposition. Using a quiz to highlight misunderstandings, the speaker demonstrates that while this notion gestures toward reality, it misleadingly inflates expectations. The discussion centers on a classic search problem—finding a unique input that causes a function to return true—and compares the runtime between classical and quantum approaches. Classically, the task scales linearly with input size $O(n)$, but quantumly, Grover's algorithm enables a square root speedup to $O(\sqrt{n})$. This represents a meaningful, though not exponential, improvement, and it is applicable to a broad class of verifiable problems known as NP.

The video builds from foundational concepts—like state vectors, qubits, and probability amplitudes—to a full geometric walkthrough of Grover's algorithm. It emphasizes that quantum computation operates not on visible parallelism, but on coherent manipulations of state vectors in high-dimensional Hilbert spaces. The real speedup arises not from evaluating all inputs at once, but from leveraging quantum reflections and interference to iteratively concentrate amplitude onto the correct output. Ultimately, the video frames quantum speedup as a kind of diagonal shortcut through a state space—a geometrically elegant and mathematically bounded detour—rather than an omnipotent parallelism. It closes with a reflection on pedagogy and analogy, including a comparison to a classical π-computing block collision problem, further grounding abstract quantum behavior in intuitive physics-inspired models.

3. Video Summary – Gradient Descent and Neural Networks

This video introduces gradient descent through the lens of training a simple neural network for digit classification using the MNIST dataset. The process begins with random initialization of over 13,000 weights and biases in a network structured with two hidden layers. The network initially performs poorly, producing output activations that don't align with the correct labels. To correct this, a cost function is defined, measuring the error between the output and the desired result by summing the squared differences across all output neurons. The average cost over thousands of training examples becomes the metric of the network's performance. Gradient descent is used to reduce this cost: by computing the slope of the cost function with respect to each parameter, the network updates its weights and biases in the direction that most quickly decreases the cost. This iterative approach, driven by backpropagation (to be discussed in a later video), forms the mathematical core of how the network "learns" by minimizing its error over time.

Throughout the video, the visual metaphor of a ball rolling down a hill is used to explain descent in both single-variable and multivariable functions. Each component of the gradient vector reflects both the direction and relative importance of adjusting each parameter. In practice, some weights have a much larger impact on classification accuracy than others, and gradient descent captures this. The video also touches on limitations: the cost landscape has many local minima, and the solution found depends on the starting point. Despite this, the basic network described achieves about 96% classification accuracy, with performance improving to 98% with minor tweaks. However, analysis of hidden layers reveals that the learned features aren't always intuitive—suggesting the network may succeed through memorization rather than generalized pattern recognition. This raises questions about structure, generalization, and overfitting in deep learning systems.[2]

[1] https://www.youtube.com/watch?v=RQWpF2Gb-gU&t=798s
[2] https://www.youtube.com/watch?v=IHZwWFHWa-w

4. Video Summary – Hash Algorithms and File Verification

This video introduces hash algorithms and their role in verifying data integrity during file transfers. The concept is compared to the check digit on barcodes or credit cards, which acts as a quick check against user error. Hash algorithms extend this concept to entire files, producing a fixed-length hexadecimal string—essentially a signature that summarizes the entire file's contents. Although it's computationally impossible to reverse a hash back to the original data, any change in the original file (even a single bit) results in a completely different hash—this property is known as the avalanche effect. Three main criteria for a good hash algorithm are identified: speed, sensitivity to small changes, and resistance to hash collisions (cases where different files produce the same hash). The video illustrates that while naturally occurring collisions are mathematically inevitable due to the pigeonhole principle, the real threat is in engineered collisions, which could be used to falsify documents while maintaining the same hash.

The video also explores real-world implications of broken hash functions. MD5, once a widely used algorithm, is now obsolete due to known vulnerabilities and hash collisions. Because it was so commonly used, and many hashes are indexed by search engines, it is trivial to reverse MD5 hashes in some cases—making it especially inappropriate for password storage. Even SHA-1, which replaced MD5, is now considered insecure due to the increasing computational power available. SHA-2 is currently the widely accepted standard, with SHA-3 being phased in. The video cautions against common misuses, like assuming hashes posted on software download sites guarantee security; if a malicious actor has compromised the server, they could alter both the software and the displayed hash. The overarching message emphasizes that while hashes are excellent for verifying integrity, they should not be treated as a full security mechanism.[3]

5. Video Summary – Cracking Enigma with Modern Methods

This video revisits the legendary Enigma machine—used by the Germans during World War II—and explores whether modern computers can efficiently break its encryption. While the Enigma's design is mechanical and was incredibly secure for its time, today's computing power invites the question: can it now be brute-forced? The video provides a clear breakdown of how the Enigma machine works: input letters pass through a plugboard, three rotors, a reflector, and back again through the system, resulting in a complex letter substitution that changes with every keystroke. This creates a dynamic cipher that was once infeasible to brute-force due to its vast number of configurations. However, the speaker attempts a ciphertext-only attack by simulating Enigma settings in code and using statistical analysis to identify likely plaintext candidates without any known plaintext assumptions.

The core method used is the index of coincidence (IOC), which quantifies how "English-like" a decrypted output is. English text typically has an IOC of around 0.067, while random strings hover near 0.038. The code tests many rotor combinations and configurations to maximize the IOC of the output, gradually approaching more intelligible plaintext. Once promising rotor configurations are identified, ring settings and plugboard combinations are iteratively optimized. Though this approach cannot crack short messages reliably—due to insufficient data for meaningful statistical patterns—longer messages (1000+ characters) yield more effective results. The final decrypted text is revealed to be a garbled but increasingly accurate version of Alan Turing's famous "Can machines think?" paper. The video concludes that although Enigma is no longer considered secure, it's not trivial to break without careful strategy. Importantly, unlike Enigma, modern encryption algorithms are resistant to partial-key brute-force attacks and do not degrade gracefully with partial key knowledge.

[3] https://www.youtube.com/watch?v=b4b8ktEV4Bg

6. Video Summary – Analog Computing and the Future of AI

This video explores the resurgence of analog computing and its potential role in the future of artificial intelligence. For centuries, analog devices were used to simulate physical systems, from calculating tides to guiding missiles. Today, a perfect storm of challenges facing digital computing—rising energy demands, the Von Neumann bottleneck, and the limitations of Moore's Law—has renewed interest in analog machines. Unlike digital computers, which operate on binary logic, analog computers represent data as continuous voltages. This allows them to simulate differential equations in real time with far fewer components and much less energy. For instance, analog circuitry can directly solve equations like the Lorenz system by physically modeling them in voltage behavior.

The connection to AI comes from the structure of neural networks, which heavily rely on matrix multiplication. Since analog circuits can multiply and sum values naturally through electrical properties, they are well-suited to AI workloads. Startups like Mythic AI have built analog chips using flash memory cells as variable resistors, enabling energy-efficient inference with neural nets. Though analog systems face issues like imprecision and noise accumulation, hybrid approaches—where analog computation is interleaved with digital processing—offer practical solutions. The video suggests that while digital has dominated computation for decades, the future might lie in combining analog's efficiency with digital's precision, especially for neural network inference. As the boundaries of computation are pushed, analog might become the key to unlocking human-like AI.[4]

7. Video Summary – Building a Minimal Programming Language

In this video, the creator walks through designing and implementing a miniature programming language from scratch using Python. The language begins as a simple reverse Polish notation (RPN) calculator that evaluates expressions like `"2 3 +"` by pushing values onto a stack and applying operations. From this basic foundation, the language is gradually expanded to support multi-line inputs, variables, and simple parsing via Python's `split()` function. The program can read from a file, tokenize each line, and maintain a variable dictionary to track assignments, enabling expressions like `"x = 2 3 +"` and `"y = x 5 +"`.

To make the system feel more like a genuine language, the video adds control flow with `while` loops and comparison operations. Loops are supported by managing a program counter and scanning lines for `while` and `end` keywords to handle conditional jumps. The culmination of the language's expressiveness is demonstrated by implementing the factorial function, showcasing variable mutation and loop-based computation. With just under 40 lines of Python code, the project illustrates how modest components—stack-based evaluation, minimal parsing, and simple control structures—can create a functioning interpreted language, providing insight into how real programming languages are built from fundamental parts.[5]

8. Video Summary – Ethics in AI, not of AI

In this Topos colloquium, philosopher and data scientist David Danks presents a compelling argument that ethics should not be confined to the deployment stage of AI, but rather integrated into every phase of research and development. Danks rejects the idea that algorithms and mathematical models are ethically neutral, asserting that choices made in problem framing, design constraints, and evaluation criteria are inherently value-laden.

Instead of relying on high-level ethical principles, which often lack actionable guidance, he advocates for practice-oriented approaches—like ethical triage checklists, model documentation tools (e.g., model cards), and value-sensitive design methodologies. These help researchers identify and navigate ethical implications throughout their work. Danks concludes by calling for a cultural shift: ethical awareness should be seen not as an optional virtue, but as an essential component of research competence.

[6]

[4] https://www.youtube.com/watch?v=GVsUOuSjvcg
[5] https://www.youtube.com/watch?v=fc1ZU8OpCCg
[6] https://www.youtube.com/watch?v=kEf_MTqeXWg

9. Video Summary – Birthday Paradox & Hash Collisions

Mike Pound introduces the birthday paradox to illustrate why hash collisions occur in cryptography. A collision happens when two distinct inputs generate the same hash output. Although hash functions, such as SHA-256, produce extremely large outputs (e.g., 2^{256} possibilities), collisions become statistically probable far sooner than intuition suggests, due to the pigeonhole principle. Mike demonstrates this paradox practically: among 23 people, the probability of a shared birthday is around 50%, and it jumps to approximately 90% with 40 people. Such collisions in cryptographic hashes can potentially weaken digital signatures and security protocols. Therefore, contemporary hash functions use significantly longer output lengths (256 bits or more) to mitigate the practical risk of collisions.

[7]

10. Video Summary – Random Numbers (the next bit)

James Clewitt explores the concept of randomness, demonstrating that people tend to choose "random" numbers predictably (e.g., many choosing 7). He distinguishes pseudorandom numbers—produced by deterministic computational algorithms—from genuinely random numbers, which stem from inherently unpredictable phenomena like radioactive decay. Clewitt uses radioactive strontium-90 and a Geiger counter to produce truly random numbers, resulting in a Gaussian distribution. He underscores that randomness is not a property of single numbers, but rather the unpredictability inherent in sequences of numbers, comparing human intuition, computational methods, and physical randomness.

[8]

5 Conclusion

Throughout CPSC-406: Algorithm Analysis, I have developed a deeper understanding of the foundational principles underlying computation, complexity, and decidability. The course began with finite automata and formal languages, laying a rigorous foundation for algorithmic thinking, and progressed through the complexities of Turing machines, undecidability, NP-completeness, and practical graph-theoretic algorithms. By working systematically through weekly exercises and reflections, I've honed my skills not only in theoretical analysis but also in effectively communicating complex concepts clearly and concisely.

Among the most valuable insights has been the direct applicability of seemingly abstract concepts, such as reducibility and complexity classes, to real-world problems like network flow optimization and satisfiability constraints in computational logic. The structured approach of documenting weekly notes, homework solutions, and explorations allowed me to synthesize my learning progressively, building both depth and breadth across topics.

In terms of improvements, future iterations of the course might benefit from further integrating algorithmic implementation with theory, offering practical coding exercises to reinforce abstract theoretical concepts more concretely. Overall, the course has significantly strengthened my analytical and critical thinking abilities, preparing me to engage more deeply in both academic research and professional software engineering contexts.

[7] https://www.youtube.com/watch?v=jsraR-el8_o
[8] https://www.youtube.com/watch?v=SxP30euw3-0

6 Bibliography

References

[HMU] J. E. Hopcroft, R. Motwani, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Archive.org Link.

[Deo] N. Deo: *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, 1974. Archive.org Link.