Automata Theory and Formal Languages
Course Report: Weeks 1-5

Student Name

May 26, 2025

Abstract

This report presents a comprehensive overview of the first five weeks of study in automata theory and formal languages. The material covers fundamental concepts including finite automata, deterministic and non-deterministic finite automata (DFA/NFA), regular languages, and state minimization techniques. Each week builds upon previous concepts, progressing from basic automaton design through advanced topics like the powerset construction and DFA minimization. The report includes detailed solutions to homework problems, practical applications, and theoretical discussions that demonstrate the relevance of automata theory to computer science and software engineering.

Contents

1 Week 1: Introduction to Finite Automata

1.1 Summary

This introductory chapter explores finite automata as fundamental computational models for processing regular languages. Finite automata are simple yet powerful machines that transition between a finite set of states based on input symbols, following predetermined rules. The chapter distinguishes between deterministic finite automata (DFA), which have exactly one transition for each state-input pair, and non-deterministic finite automata (NFA), which may have multiple possible transitions.

A practical example demonstrates an electronic money system where automata model transactions between customers, stores, and banks. The system governs five critical operations: paying, canceling, shipping, redeeming, and transferring money. Each entity follows strict transition rules to ensure transaction validity and prevent fraud, such as preventing reuse of digital money files.

The chapter also introduces error handling through self-loops for irrelevant inputs, allowing automata to remain functional when encountering unexpected behavior. Product automata are presented as a technique for combining multiple automata into larger systems, enabling comprehensive error-checking across multiple interacting components.

The analysis extends to identifying reachable states and failure points, such as attempting to cancel already-processed payments, highlighting the importance of careful automaton design in real-world applications.

1.2 Homework Solutions

1.2.1 Problem 1: Parking/Vending Machine

Problem: Characterize all accepted words for the parking/vending machine automaton.

Solution:

The automaton has:

- States: $\{0, 5, 10, 15, 20, 25\}$ representing accumulated payment
- Alphabet: $\{5, 10\}$ representing coin denominations
- Initial state: 0
- Final state: 25

Transitions follow: From state s, on input c: go to state $\min(s + c, 25)$

Characterization: A word is accepted if and only if the sum of all coins equals at least 25 cents.

Formally: $L = \{w \in \{5, 10\}^* \mid \sum_{c \in w} c \geq 25\}$

1.2.2 Problem 2: Turnstile Automaton

Problem: Characterize all accepted words for the turnstile automaton.

Solution:

The turnstile has states $\{\text{locked}, \text{unlocked}\}$ with transitions:

- locked $\xrightarrow{\text{pay}}$ unlocked
- locked $\xrightarrow{\text{push}}$ locked
- unlocked $\xrightarrow{\text{pay}}$ unlocked
- unlocked $\xrightarrow{\text{push}}$ locked

Characterization: A word is accepted if it ends in the unlocked state, which occurs when the number of "pay" actions exceeds the number of "push" actions by an odd amount.

Regular Expression: $(\text{push}^* \cdot \text{pay} \cdot (\text{pay}^2 + \text{push}^2)^*)$

1.2.3 Problem 3: DFA Word Analysis

Problem: Determine which words $w_1 = 0010$, $w_2 = 1101$, $w_3 = 1100$ end in the accepting state.

Solution:

Tracing through the given DFA:

For $w_1 = 0010$: $q_0 \xrightarrow{0} q_2 \xrightarrow{0} q_2 \xrightarrow{1} q_1 \xrightarrow{0} q_1$

For $w_2 = 1101$: $q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_2 \xrightarrow{1} q_1$

For $w_3 = 1100$: $q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_2 \xrightarrow{0} q_2$

Words w_1 and w_2 are accepted; w_3 is rejected.

1.2.4 Problem 4: Language Membership

Problem: Determine membership in languages:

- $L_1 := \{x01y \mid x, y \in \Sigma^*\}$ (contains substring "01")
- $L_2 := \{w \mid |w| = 2^n \text{ for some } n \in \mathbb{N}\}$ (length is power of 2)
- $L_3 := \{w \mid |w|_0 = |w|_1\}$ (equal numbers of 0s and 1s)

Solution:

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

1.3 Questions for Further Study

1. How can automata theory be applied to model and verify communication protocols in distributed systems, and what are the limitations of finite-state models in representing complex network behaviors?
2. In what ways do the principles of automaton design influence the architecture of lexical analyzers in modern compilers, and how does state minimization impact parsing performance?

2 Week 2: DFA Operations and Constructions

2.1 Summary

Week 2 focuses on fundamental operations with deterministic finite automata and their implementation. The material covers DFA simulation through the run method, language characterization, and Boolean operations on regular languages including complement and intersection.

Key topics include implementing DFA execution algorithms, designing automata for specific language patterns, and understanding closure properties of regular languages. The complement construction demonstrates that regular languages are closed under complementation by simply inverting the set of final states. The intersection construction uses the Cartesian product of state sets to create automata that accept words belonging to both input languages.

The practical component involves Python implementation of DFA operations, bridging theoretical concepts with software engineering practices. This reinforces understanding of transition functions, error handling, and algorithmic efficiency in automaton simulation.

2.2 Exercise Solutions

2.2.1 Exercise 1: Word Processing with DFAs

1.1 Acceptance Table:

w	accepted by A_1?	accepted by A_2?
aaa	F	T
aab	T	F
aba	F	F
abb	T	F
baa	F	T
bab	F	F
bba	F	F
bbb	F	F

1.2 Languages:

$$L(A_1) = \{w \in \{a,b\}^* : w \text{ starts with } a \text{ and ends with } b\}$$
$$L(A_2) = \{w \in \{a,b\}^* : w \text{ ends with } aa\}$$

2.2.2 Exercise 4: Complement Automaton

For DFA $A = (Q, \Sigma, \delta, q_0, F)$, the complement automaton is:

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

Python Implementation:

```
def complement(self):
    return DFA(
        Q=self.Q,
        Sigma=self.Sigma,
        delta=self.delta,
        q0=self.q0,
        F=set(self.Q) - set(self.F)
    )
```

2.2.3 Exercise 5: Intersection Automaton

For DFAs A and B, the intersection automaton C has:

$$Q_C = Q_A \times Q_B$$
$$q_{0,C} = (q_{0,A}, q_{0,B})$$
$$F_C = F_A \times F_B$$
$$\delta_C((p,q),a) = (\delta_A(p,a), \delta_B(q,a))$$

2.3 Questions for Further Study

1. How do the closure properties of regular languages under Boolean operations enable the construction of complex pattern-matching systems in text processing applications?
2. What are the computational complexity implications of DFA intersection and union operations in the context of model checking and formal verification systems?

3 Week 3: Extended Transitions and Product Automata

3.1 Summary

Week 3 deepens understanding of automata theory through extended transition functions and product constructions. The extended transition function $\hat{\delta}$ formalizes multi-step transitions from states through entire strings, providing the mathematical foundation for automaton acceptance.

Product automata construction enables combining multiple automata to model complex systems with multiple constraints. The intersection construction creates automata accepting languages that satisfy all component specifications simultaneously, while union construction accepts languages satisfying any component specification.

Structural induction on string length emerges as a fundamental proof technique, demonstrated through formal verification of transition properties. This mathematical rigor underpins correctness arguments in compiler design and system verification.

3.2 Homework Solutions

3.2.1 Homework 1: Extended Transition Function

Problem 1: $L(\mathcal{A}^{(2)})$ accepts words of odd length where every odd-positioned letter is 'a':

$$L(\mathcal{A}^{(2)}) = \{w \in \{a,b\}^* : |w| \equiv 1 \pmod 2, w_{2i+1} = a\}$$

Problem 2: Computing extended transitions:
For $\hat{\delta}^{(1)}(1, abaa)$:

$$\begin{aligned}\hat{\delta}^{(1)}(1,\varepsilon) &= 1\\ \hat{\delta}^{(1)}(1,a) &= 2\\ \hat{\delta}^{(1)}(1,ab) &= 3\\ \hat{\delta}^{(1)}(1,aba) &= 2\\ \hat{\delta}^{(1)}(1,abaa) &= 2\end{aligned}$$

3.2.2 Homework 2: Product Automaton

The intersection automaton has:

$$\begin{aligned}Q &= Q_1 \times Q_2\\ q_0 &= (q_0^{(1)}, q_0^{(2)})\\ F &= F^{(1)} \times F^{(2)}\\ \delta((p,q),a) &= (\delta^{(1)}(p,a), \delta^{(2)}(q,a))\end{aligned}$$

Proof: A word w is accepted iff its run ends in $(p,q) \in F^{(1)} \times F^{(2)}$, which occurs exactly when w is accepted by both component automata.

3.3 Induction Proof Example

Exercise 2.2.7: For state q with $\delta(q,a) = q$ for all $a \in \Sigma$, prove $\hat{\delta}(q,w) = q$ for all $w \in \Sigma^*$.

Proof by induction on $|w|$:

Base case: $|w| = 0$, so $w = \varepsilon$. By definition, $\hat{\delta}(q,\varepsilon) = q$.

Inductive step: Assume the claim holds for all words of length n. For $|w| = n+1$, write $w = xa$ where $|x| = n$ and $a \in \Sigma$:

$$\hat{\delta}(q,xa) = \delta(\hat{\delta}(q,x),a) = \delta(q,a) = q$$

The second equality uses the inductive hypothesis, and the third uses the loop property.

3.4 Questions for Further Study

1. How does the concept of extended transition functions generalize to pushdown automata and Turing machines, and what additional mathematical structures are required?

2. In what ways can product automaton constructions be optimized to avoid state explosion in practical model checking applications?

4 Week 4: Nondeterministic Finite Automata

4.1 Summary

Week 4 introduces nondeterministic finite automata (NFAs) and establishes their equivalence with DFAs through the powerset construction. NFAs allow multiple transitions from a single state on the same input symbol, enabling more natural modeling of certain computational processes.

The key insight is that despite their apparent additional power, NFAs recognize exactly the same class of languages as DFAs—the regular languages. The powerset construction provides an algorithmic transformation from any NFA to an equivalent DFA, though potentially with exponential state growth.

This equivalence has profound implications for language theory and practical applications, as it allows leveraging the conceptual simplicity of NFAs for design while maintaining the algorithmic predictability of DFAs for implementation.

4.2 Homework Solutions

4.2.1 Homework 1: Viewing DFA as NFA

Any DFA $A = (Q, \Sigma, \delta, q_0, F)$ can be viewed as an NFA by redefining the transition function:

$$\delta_N(q, a) = \{\delta(q, a)\}$$

This creates singleton sets for each transition, maintaining deterministic behavior while conforming to NFA notation. Clearly $L(A) = L(A')$ since no additional nondeterminism is introduced.

4.2.2 Homework 2: Powerset Construction

For NFA $N = (Q, \Sigma, \delta_N, q_0, F)$, the equivalent DFA is $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ where:

- $Q_D = \mathcal{P}(Q)$ (powerset of Q)
- $\delta_D(S, a) = \bigcup_{q \in S} \delta_N(q, a)$
- $F_D = \{S \subseteq Q : S \cap F \neq \emptyset\}$

Example Construction: Starting from $\{q_0\}$, we systematically compute all reachable subsets and their transitions, creating a DFA that simulates all possible NFA computations simultaneously.

4.3 Questions for Further Study

1. How do modern regular expression engines balance the use of NFA and DFA representations to optimize both compile-time and runtime performance?

2. What techniques can mitigate the exponential state explosion in powerset construction when dealing with large NFAs derived from complex regular expressions in practical applications?

5 Week 5: Regular Expressions and DFA Minimization

5.1 Summary

Week 5 completes the foundation of regular language theory by connecting DFAs to regular expressions and introducing state minimization techniques. The state elimination method provides a systematic algorithm for converting any DFA to an equivalent regular expression, demonstrating that every DFA-recognizable language has a finite regular expression representation.

DFA minimization addresses the practical problem of reducing automaton size while preserving language recognition. The table-filling algorithm systematically identifies distinguishable state pairs, enabling construction of minimal DFAs with the fewest possible states. This optimization is crucial for memory-efficient implementations in lexical analyzers and pattern-matching systems.

The relationship between regular expressions, NFAs, and DFAs forms the theoretical foundation for modern text processing tools, compiler front-ends, and formal verification systems.

5.2 Exercise Solutions

5.2.1 Exercise 3.2.1: State Elimination Method

Given the DFA transition table, we apply the state elimination algorithm to derive regular expressions $R_{ij}^{(k)}$ representing paths from state i to state j using intermediate states $1, 2, \ldots, k$.

Base Case ($R_{ij}^{(0)}$): Direct transitions without intermediate states:

$$R_{11}^{(0)} = 1, \quad R_{12}^{(0)} = 0, \quad R_{13}^{(0)} = \emptyset \tag{1}$$

$$R_{21}^{(0)} = 1, \quad R_{22}^{(0)} = \emptyset, \quad R_{23}^{(0)} = 0 \tag{2}$$

$$R_{31}^{(0)} = \emptyset, \quad R_{32}^{(0)} = 1, \quad R_{33}^{(0)} = 0 \tag{3}$$

Using the recurrence $R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$:

After eliminating state 1: $R_{11}^{(1)} = 1^*$, $R_{12}^{(1)} = 1^*0$, etc.

Final regular expression: The language accepted by the automaton is:

$$L = R_{13}^{(3)} = 1^*0(1^*0 + 0)^*$$

5.2.2 Exercise 4.4.1: DFA Minimization

The table-filling algorithm identifies distinguishable state pairs:

Step 1: Mark all pairs (p, q) where exactly one of p, q is accepting.

Step 2: Iteratively mark pairs (p, q) if there exists input a such that $(\delta(p, a), \delta(q, a))$ is already marked.

Result: After systematic analysis, states can be partitioned into equivalence classes. The minimized DFA combines equivalent states while preserving the original language.

For this particular DFA, the analysis reveals that the automaton reduces to 2 states: one equivalence class containing all non-accepting states, and another containing the single accepting state D.

5.2.3 Exercise 4.4.2: Advanced Minimization

This exercise demonstrates a DFA that is already minimal—all 9 states are pairwise distinguishable. The systematic application of the distinguishability test confirms that no further state reduction is possible without changing the recognized language.

5.3 Questions for Further Study

1. How does the computational complexity of DFA minimization scale with automaton size, and what approximation algorithms exist for large-scale applications where exact minimization is computationally prohibitive?

2. In what ways do state minimization techniques for DFAs extend to other computational models like pushdown automata or finite-state transducers used in natural language processing?

6 Conclusion

The first five weeks of automata theory study establish a solid foundation in finite-state computation and regular languages. Beginning with basic automaton concepts and progressing through advanced topics like nondeterminism and minimization, the material demonstrates both theoretical elegance and practical relevance.

Key achievements include:

- Mastery of DFA and NFA design for pattern recognition
- Understanding of closure properties and Boolean operations on regular languages
- Proficiency in automaton transformations (NFA to DFA conversion, minimization)
- Connection between automata and regular expressions
- Appreciation for applications in compiler design and text processing

These foundational concepts prepare for advanced topics in formal language theory, including context-free languages, pushdown automata, and Turing machines. The mathematical rigor developed through inductive proofs and construction algorithms provides essential skills for theoretical computer science and software verification.

The practical implementations and real-world examples demonstrate how abstract mathematical models translate into concrete computational tools, reinforcing the relevance of theoretical computer science to software engineering practice.

7 Week 6 & 7: Turing Machines and Computational Theory

7.1 Summary

Turing machines (TMs) form the foundation of modern computation theory. They extend deterministic finite automata (DFAs) by incorporating features like an infinite tape, the ability to read and write, and movement in both directions. While DFAs are limited to recognizing regular languages, TMs are powerful enough to perform arithmetic and recognize more complex, non-regular languages.

A DFA can be simulated by a TM by treating it as a read-only device that scans input from left to right and halts at the end. However, the DFA cannot store unbounded information or perform arithmetic operations. This is why languages like

$$L = \{10^n 10^m 10^{m+n} : n, m \in \mathbb{N}\}$$

are not regular—DFAs lack the memory needed to compare arbitrary counts.

Alan Turing introduced the TM model to formalize computation. He defined a TM as a 7-tuple containing a set of states, input symbols, tape symbols, a transition function, start and accept states, and a blank symbol. The TM reads one symbol, writes another, moves the head, and changes state based on the transition function.

Turing machines can accept inputs by entering an accepting state. The sequence of configurations the machine goes through is captured in its *Instantaneous Descriptions* (IDs), which show the tape contents and current state. Transition diagrams visually represent TM behavior and help in designing machines.

Advanced exercises involve constructing TMs to recognize non-regular languages (like $\{0^n 1^n\}$), perform arithmetic (unary addition, string doubling), and manipulate strings (swapping characters or reversing input). Multitape TMs extend the model by using multiple tapes for efficiency, though they are computationally equivalent to single-tape TMs.

Languages are classified based on how TMs process them. A language is **recursively enumerable (RE)** if there exists a TM that accepts it, possibly without halting for non-members. A language is **recursive** (or **decidable**) if a TM always halts and correctly accepts or rejects any input.

Some problems, such as the halting problem (whether a TM halts on a given input), are undecidable—there is no TM that can solve them for all inputs. This gives rise to distinctions among decidable, RE, and co-RE (complement of RE) languages. Finally, *Turing completeness* is the property of a system that can simulate any TM, establishing the limits of what can be computed.

7.2 Homework Solutions

7.2.1 Exercise A

1. TM to accept $L = \{10^n : n \in \mathbb{N}\}$ and return 10^{n+1} for input 10^n:
This Turing Machine needs to:

- Accept strings of the form 10^n where $n \geq 0$
- Transform the input by adding one more 0 at the end

States: $\{q_0, q_1, q_2, q_3, q_{accept}, q_{reject}\}$
Transition function:

$$\delta(q_0, 1) = (q_1, 1, R) \quad \text{(read the initial 1)} \tag{4}$$
$$\delta(q_0, 0) = (q_{reject}, 0, R) \quad \text{(reject if starts with 0)} \tag{5}$$
$$\delta(q_0, \text{blank}) = (q_{reject}, \text{blank}, R) \quad \text{(reject empty string)} \tag{6}$$
$$\delta(q_1, 0) = (q_1, 0, R) \quad \text{(scan through 0's)} \tag{7}$$
$$\delta(q_1, \text{blank}) = (q_2, 0, L) \quad \text{(add extra 0 and go back)} \tag{8}$$
$$\delta(q_2, 0) = (q_2, 0, L) \quad \text{(go back to start)} \tag{9}$$
$$\delta(q_2, 1) = (q_{accept}, 1, R) \quad \text{(accept)} \tag{10}$$

2. TM to accept $L = \{10^n : n \in \mathbb{N}\}$ and return string 1 for input 10^n:
This Turing Machine needs to:

- Accept strings of the form 10^n where $n \geq 0$
- Replace the entire input with just the string 1

States: $\{q_0, q_1, q_2, q_3, q_{accept}, q_{reject}\}$
Transition function:

$$\delta(q_0, 1) = (q_1, 1, R) \quad \text{(read the initial 1)} \tag{11}$$
$$\delta(q_0, 0) = (q_{reject}, 0, R) \quad \text{(reject if starts with 0)} \tag{12}$$
$$\delta(q_0, \text{blank}) = (q_{reject}, \text{blank}, R) \quad \text{(reject empty string)} \tag{13}$$
$$\delta(q_1, 0) = (q_1, \text{blank}, R) \quad \text{(erase 0's)} \tag{14}$$
$$\delta(q_1, \text{blank}) = (q_2, \text{blank}, L) \quad \text{(go back to start)} \tag{15}$$
$$\delta(q_2, \text{blank}) = (q_2, \text{blank}, L) \quad \text{(continue going left)} \tag{16}$$
$$\delta(q_2, 1) = (q_{accept}, 1, R) \quad \text{(accept at the 1)} \tag{17}$$

3. TM to accept all binary strings and swap 0's and 1's:
This Turing Machine transforms every binary string by swapping all 0's to 1's and all 1's to 0's.
States: $\{q_0, q_1, q_{accept}\}$

Transition function:

$$\delta(q_0, 0) = (q_0, 1, R) \quad \text{(change 0 to 1)} \tag{18}$$
$$\delta(q_0, 1) = (q_0, 0, R) \quad \text{(change 1 to 0)} \tag{19}$$
$$\delta(q_0, \text{blank}) = (q_1, \text{blank}, L) \quad \text{(go back to start)} \tag{20}$$
$$\delta(q_1, 0) = (q_1, 0, L) \quad \text{(move left)} \tag{21}$$
$$\delta(q_1, 1) = (q_1, 1, L) \quad \text{(move left)} \tag{22}$$
$$\delta(q_1, \text{blank}) = (q_{accept}, \text{blank}, R) \quad \text{(accept)} \tag{23}$$

7.2.2 Exercise 1: Language Classification

Classify the following languages as decidable, recursively enumerable (r.e.), or having recursively enumerable complement (co-r.e.):

1. $L_1 := \{M | M$ **halts on itself**$\}$
This is the **Halting Problem** for machines on themselves.

- L_1 is **recursively enumerable (r.e.)** but not decidable
- We can enumerate L_1: simulate M on input $\langle M \rangle$ and accept if it halts
- $\overline{L_1}$ is **not recursively enumerable**
- By Rice's theorem and the undecidability of the halting problem, L_1 is not decidable

Classification: r.e. but not decidable

2. $L_2 := \{(M, w) | M$ **halts on the word** $w\}$
This is the general **Halting Problem**.

- L_2 is **recursively enumerable (r.e.)** but not decidable
- We can enumerate L_2: simulate M on input w and accept if it halts
- $\overline{L_2}$ is **not recursively enumerable**
- This is the classic undecidable problem

Classification: r.e. but not decidable

3. $L_3 := \{(M, w, k) | M$ **halts on** w **in at most** k **steps**$\}$
This is a **bounded halting problem**.

- L_3 is **decidable**
- Algorithm: Simulate M on w for exactly k steps. If it halts within k steps, accept; otherwise reject
- Since we only simulate for a finite number of steps, the algorithm always terminates

Classification: decidable

7.2.3 Exercise 2: Properties of Decidable and R.E. Languages

Determine if the following statements are true or false:

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

TRUE. Proof: If L_1 and L_2 are decidable, then there exist Turing machines M_1 and M_2 that decide them respectively. We can construct a Turing machine M that decides $L_1 \cup L_2$ as follows: On input w:

1. Run M_1 on w
2. If M_1 accepts, then accept
3. If M_1 rejects, run M_2 on w

4. If M_2 accepts, then accept; if M_2 rejects, then reject

Since both M_1 and M_2 always halt, M always halts and correctly decides $L_1 \cup L_2$.

2. If L is decidable, then so is the complement $\overline{L} := \Sigma^* \setminus L = \{w \in \Sigma^* | w \notin L\}$.

TRUE. Proof: If L is decidable, then there exists a Turing machine M that decides L. We can construct a Turing machine M' that decides $\overline{L}$ as follows: On input w:

1. Run M on w
2. If M accepts, then reject
3. If M rejects, then accept

Since M always halts, M' always halts and correctly decides $\overline{L}$.

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

TRUE. Proof: If L is decidable, then L^* is also decidable. We can use dynamic programming to check if a string w can be decomposed into strings from L. The algorithm works as follows: On input w of length n:

1. Create a boolean array $dp[0..n]$ where $dp[i]$ indicates if $w[0..i-1]$ is in L^*
2. Set $dp[0] = true$ (empty string is in L^*)
3. For $i = 1$ to n:
   - Set $dp[i] = false$
   - For $j = 0$ to $i - 1$:
     - If $dp[j] = true$ and $w[j..i-1] \in L$, then set $dp[i] = true$ and break
4. Return $dp[n]$

Since L is decidable, we can check membership in L in finite time, making this algorithm decidable.

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

TRUE. Proof: If L_1 and L_2 are r.e., then there exist Turing machines M_1 and M_2 that recognize them respectively. We can construct a Turing machine M that recognizes $L_1 \cup L_2$ using dovetailing: On input w:

1. Simulate M_1 and M_2 on w in parallel (dovetailing)
2. If either M_1 or M_2 accepts, then accept
3. If both reject (when they halt), then reject

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

FALSE. Counterexample: Let L be the halting problem $\{(M, w) | M \text{ halts on } w\}$. We know that L is r.e. but $\overline{L}$ is not r.e. If both L and $\overline{L}$ were r.e., then L would be decidable (since we could run both recognizers in parallel and one would eventually accept). But the halting problem is undecidable, so $\overline{L}$ cannot be r.e.

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

TRUE. Proof: If L is r.e., then there exists a Turing machine M that recognizes L. We can construct a Turing machine M' that recognizes L^* as follows: On input w:

1. If $w = \epsilon$, accept (empty string is in L^*)
2. Non-deterministically guess a decomposition of w into $w_1 w_2 ... w_k$ where each $w_i \neq \epsilon$
3. For each w_i, simulate M on w_i
4. If all simulations accept, then accept
5. If any simulation rejects, try another decomposition

Since we can enumerate all possible decompositions and M will eventually accept strings in L, this procedure will recognize L^*.

7.3 Questions

Question 1: How does the concept of Turing completeness help us understand the limits of what can be computed by programming languages, and how does it relate to the halting problem?

Question 2: Why can't deterministic finite automata (DFAs) recognize languages like $L = \{0^n 1^n : n \in \mathbb{N}\}$, and how do Turing machines overcome this limitation to perform arithmetic or simulate memory?

8 Week 8 & 9: Big O Notation and Complexity Analysis

8.1 Summary

Big O notation, also known as Landau notation, is a mathematical tool used to describe the upper bound of a function's growth rate. While its technical definition involves bounding a function $f(n)$ by another function $g(n)$ up to a constant factor beyond some threshold N, its true power lies in its metaphorical use in computer science—representing performance, scalability, and complexity.

The notes begin by framing Big O as more than just a formal definition; it is a metaphor essential for understanding the efficiency of algorithms and the feasibility of solving problems with limited resources. A warmup exercise helps build intuition by comparing the growth rates of common mathematical functions, such as logarithmic, polynomial, exponential, and factorial.

From there, the formal definition is introduced: $f(n) \in \mathcal{O}(g(n))$ means that for large enough n, $f(n) \leq M \cdot g(n)$. The properties of Big O, such as closure under addition, scaling by constants, and asymptotic comparison, form a toolkit for analyzing algorithmic behavior.

Using these properties, we can compare algorithms by their runtime complexity. Exercises demonstrate how sorting algorithms like bubble sort, merge sort, and quicksort differ significantly in Big O terms. These comparisons are vital for selecting efficient solutions in practice.

Ultimately, Big O allows software engineers to classify problems and algorithms by resource usage, especially time and space, making it a cornerstone of algorithm analysis and design.

8.2 Homework Solutions

8.2.1 Exercise 1: Order of Growth (Slow to Fast)

Order the functions by their asymptotic growth:

$$\log(\log n) < \log n < e^{\log n} (= n) < e^{2 \log n} (= n^2) < 2^n < e^n < n!$$

8.2.2 Exercise 2: Basic Big O Properties

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

1. $f \in \mathcal{O}(f)$ - **Reflexivity**
2. If $c > 0$, then $\mathcal{O}(c \cdot f) = \mathcal{O}(f)$ - **Constant scaling**
3. If $f \leq g$ for large inputs, then $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ - **Monotonicity**
4. If $\mathcal{O}(f) \subseteq \mathcal{O}(g)$, then $\mathcal{O}(f + h) \subseteq \mathcal{O}(g + h)$ - **Additive property**
5. If $h(n) > 0$ and $\mathcal{O}(f) \subseteq \mathcal{O}(g)$, then $\mathcal{O}(f \cdot h) \subseteq \mathcal{O}(g \cdot h)$ - **Multiplicative property**

8.2.3 Exercise 3: Useful Asymptotic Bounds

Let $i, j, k, n \in \mathbb{N}$. Then:

1. If $j \leq k$, then $\mathcal{O}(n^j) \subseteq \mathcal{O}(n^k)$
2. $\mathcal{O}(n^j + n^k) \subseteq \mathcal{O}(n^k)$ (assuming $k \geq j$)

3. $\mathcal{O}\left(\sum_{i=0}^{k} a_i n^i\right) = \mathcal{O}(n^k)$
4. $\mathcal{O}(\log n) \subseteq \mathcal{O}(n)$
5. $\mathcal{O}(n \log n) \subseteq \mathcal{O}(n^2)$

8.2.4 Exercise 4: Growth Function Comparisons

1. $\mathcal{O}(n) \supseteq \mathcal{O}(\sqrt{n})$ - **TRUE**
2. $\mathcal{O}(n^2) \subset \mathcal{O}(2^n)$ - **TRUE**
3. $\mathcal{O}(\log n) \subseteq \mathcal{O}(\log n \cdot \log n)$ - **TRUE**
4. $\mathcal{O}(2^n) \subset \mathcal{O}(3^n)$ - **TRUE**
5. $\mathcal{O}(\log_2 n) = \mathcal{O}(\log_3 n)$ - **TRUE**

8.2.5 Exercise 5: Sorting Algorithm Comparisons

Algorithm A	Algorithm B	Comparison
Bubble Sort	Insertion Sort	Both $O(n^2)$; insertion better on nearly sorted input
Insertion Sort	Merge Sort	$O(n^2)$ vs $O(n \log n)$; merge sort better
Merge Sort	Quick Sort	Both average-case $O(n \log n)$, worst-case quick sort is $O(n^2)$

8.3 Questions

Question 1: How does Big O analysis influence the choice of algorithms in real-world software systems, especially when working with large datasets or high-performance requirements?

Question 2: In what ways can misconceptions about Big O (e.g., ignoring constant factors or worst-case focus) lead to inefficient or inappropriate engineering decisions?

9 Week 10/11: Boolean Logic and SAT Problems

9.1 Summary

This week focused on Boolean satisfiability (SAT) problems, a fundamental concept in computer science and computational complexity theory. We explored converting Boolean formulas to Conjunctive Normal Form (CNF), determining satisfiability of logical expressions, and encoding complex constraint problems like Sudoku using Boolean variables. These techniques are essential for automated reasoning, verification, and solving NP-complete problems.

9.2 Homework

9.2.1 Exercise 1: Converting to CNF

Problem: Rewrite the following formulas in CNF:

1. $\varphi_1 := \neg((a \wedge b) \vee (\neg c \wedge d))$
2. $\varphi_2 := \neg((p \vee q) \to (r \wedge \neg s))$

Recall that $a \to b := \neg a \vee b$.
Solution:

For φ_1:

$$\varphi_1 = \neg((a \wedge b) \vee (\neg c \wedge d)) \tag{24}$$
$$= \neg(a \wedge b) \wedge \neg(\neg c \wedge d) \quad \text{(De Morgan's law)} \tag{25}$$
$$= (\neg a \vee \neg b) \wedge (c \vee \neg d) \quad \text{(De Morgan's law)} \tag{26}$$

This is already in CNF: $(\neg a \vee \neg b) \wedge (c \vee \neg d)$
For φ_2:

$$\varphi_2 = \neg((p \vee q) \to (r \wedge \neg s)) \tag{27}$$
$$= \neg(\neg(p \vee q) \vee (r \wedge \neg s)) \quad \text{(Definition of implication)} \tag{28}$$
$$= \neg\neg(p \vee q) \wedge \neg(r \wedge \neg s) \quad \text{(De Morgan's law)} \tag{29}$$
$$= (p \vee q) \wedge (\neg r \vee s) \quad \text{(Double negation and De Morgan's law)} \tag{30}$$

This is already in CNF: $(p \vee q) \wedge (\neg r \vee s)$

9.2.2 Exercise 2: Satisfiability

Problem: Are the following formulas satisfiable? For each one, if yes, give an assignment that makes it true; if not, explain why there do not exist any.

1. $\psi_1 := (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$
2. $\psi_2 := (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg p \vee r)$
3. $\psi_3 := (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$

Solution:
For ψ_1: Let's analyze the constraints:

- $(a \vee \neg b)$: Either a is true or b is false
- $(\neg a \vee b)$: Either a is false or b is true
- $(\neg a \vee \neg b)$: Either a is false or b is false

From clauses 1 and 2: If a is true, then b must be true. If a is false, then b can be anything. From clause 3: At least one of a or b must be false.

If $a = \text{true}$, then $b = \text{true}$ (from clauses 1,2), but this violates clause 3. If $a = \text{false}$, then clause 2 is satisfied, clause 3 requires $b = \text{false}$, and clause 1 requires $b = \text{false}$.

Answer: ψ_1 is satisfiable with assignment $a = \text{false}, b = \text{false}$.
For ψ_2: Answer: ψ_2 is satisfiable. One satisfying assignment is $p = \text{false}, q = \text{true}, r = \text{true}$.
Verification:

- $(\neg p \vee q) = (\text{true} \vee \text{true}) = \text{true}$
- $(\neg q \vee r) = (\text{false} \vee \text{true}) = \text{true}$
- $(\neg p \vee r) = (\text{true} \vee \text{true}) = \text{true}$

For ψ_3: Let's consider all possible assignments for x and y:

- If $x = \text{true}, y = \text{true}$: $(\neg x \vee \neg y)$ becomes false
- If $x = \text{true}, y = \text{false}$: $(\neg x \vee y)$ becomes false
- If $x = \text{false}, y = \text{true}$: $(x \vee \neg y)$ becomes false
- If $x = \text{false}, y = \text{false}$: $(x \vee y)$ becomes false

Answer: ψ_3 is unsatisfiable because every possible truth assignment makes at least one clause false.

9.2.3 Exercise 3: Encoding Sudoku

Problem: Write boolean formulas encoding the rules of Sudoku. For each row $r \in \{1, \ldots, 9\}$, column $c \in \{1, \ldots, 9\}$, and value $v \in \{1, \ldots, 9\}$ we introduce a variable $x_{r,c,v}$. The intuition is that $x_{r,c,v}$ is true if and only if the entry (r, c) has value v.

Solution:

The six conditions can be formalized as follows:

C_1**: "each entry has at least one value"**

$$C_1 := \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \bigvee_{v=1}^{9} x_{r,c,v}$$

C_2**: "each entry has at most one value"**

$$C_2 := \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{8} \bigwedge_{v'=v+1}^{9} (\neg x_{r,c,v} \vee \neg x_{r,c,v'})$$

C_3**: "each row has all numbers"**

$$C_3 := \bigwedge_{r=1}^{9} \bigwedge_{v=1}^{9} \bigvee_{c=1}^{9} x_{r,c,v}$$

C_4**: "each column has all numbers"**

$$C_4 := \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{9} \bigvee_{r=1}^{9} x_{r,c,v}$$

C_5**: "each** (3×3)**-block has all numbers"**

$$C_5 := \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{v=1}^{9} \bigvee_{r=3i+1}^{3i+3} \bigvee_{c=3j+1}^{3j+3} x_{r,c,v}$$

C_6**: "the solution respects the given clues"** For each given clue where position (r_0, c_0) has value v_0:

$$C_6 := \bigwedge_{\text{clues } (r_0,c_0,v_0)} x_{r_0,c_0,v_0}$$

The complete Sudoku constraint is:

$$\varphi := C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$$

Note: We can also add uniqueness constraints for rows, columns, and blocks (ensuring each number appears at most once), but these are often implied by the combination of the "at least one" and "at most one" constraints above.

9.3 Questions

1. **Complexity Theory:** Given that SAT is NP-complete, what does this imply about the computational complexity of solving Sudoku puzzles? Explain why Sudoku solving algorithms might still be practical despite this theoretical hardness.

2. **Boolean Satisfiability:** Describe how the DPLL algorithm would approach solving the unsatisfiable formula ψ_3 from Exercise 2. What optimizations (like unit propagation or pure literal elimination) could speed up the detection of unsatisfiability?

10 Week 12/13: Network Flow and Algorithm Analysis

10.1 Summary

This week covered network flow algorithms and algorithm analysis techniques. We studied the Ford-Fulkerson algorithm for finding maximum flows in networks, explored the max-flow min-cut theorem, and analyzed the time complexity of nested loop algorithms. These concepts are fundamental to optimization problems, resource allocation, and understanding algorithmic efficiency in computer science.

10.2 Homework

10.2.1 Exercise 1: Maximal Flow & Minimal Cut

Part 1: Ford-Fulkerson Algorithm Given the flow network G with capacities shown as flow/capacity on each edge, we apply the Ford-Fulkerson algorithm to find the maximal flow.

Initial Network: The network has the following edges with capacities:

- $s \to a$: capacity 10, current flow 8
- $s \to b$: capacity 10, current flow 0
- $a \to c$: capacity 4, current flow 0
- $a \to b$: capacity 2, current flow 0
- $a \to d$: capacity 8, current flow 8
- $b \to d$: capacity 9, current flow 0
- $c \to t$: capacity 10, current flow 0
- $c \to d$: capacity 6, current flow 0
- $d \to t$: capacity 10, current flow 8

Step-by-step Ford-Fulkerson execution:

Looking at the network systematically, we need to find all possible augmenting paths and their bottleneck capacities.

Analysis: Starting with the current flow of 8, we can identify several possible augmenting paths:

- Path $s \to b \to d \to t$: bottleneck $= \min(10, 9, 2) = 2$
- Path $s \to a \to c \to t$: bottleneck $= \min(2, 4, 10) = 2$
- Path $s \to b \to d \to c \to t$: bottleneck $= \min(8, 7, 6, 10) = 6$

After careful analysis of all possible flow combinations and the constraint imposed by edge capacities, the maximum achievable flow is limited by the minimum cut.

Maximum Flow: 19

Part 2: Minimal Cut To find the minimal cut, we need to identify the cut that separates source s from sink t with minimum total capacity.

Analyzing possible cuts:

- Cut $\{s\}|\{a, b, c, d, t\}$: capacity $= 10 + 10 = 20$
- Cut $\{s, b\}|\{a, c, d, t\}$: capacity $= 10 + 9 = 19$
- Cut $\{s, a, b, c, d\}|\{t\}$: capacity $= 10 + 10 = 20$

Minimum Cut: The cut $\{s, b\}|\{a, c, d, t\}$ with edges (s, a) and (b, d) having total capacity $10 + 9 = 19$.

By the max-flow min-cut theorem, this confirms our maximum flow value of 19.

Part 3: Uniqueness of Maximum Flow **Answer:** No, the maximal flow is not necessarily unique.

Explanation: While the *value* of the maximum flow is always unique (by the max-flow min-cut theorem), the actual flow assignment on individual edges can vary. There can be multiple ways to achieve the same maximum flow value by routing flow through different paths in the network.

For example, if there are multiple edge-disjoint paths from source to sink, we might distribute the flow differently among these paths while maintaining the same total flow value.

10.2.2 Exercise 2: Algorithm Analysis

Part 1: Return Value Analysis Given 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)
```

Let's trace through what this algorithm counts:
For each k from 1 to $n-1$:

- For each l from $k+1$ to n:
  - For each m from 1 to l: increment r
  - This adds l to r for each valid l

So we're computing:

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

Using the hint that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$:

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

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

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

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

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

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

After algebraic simplification:

$$r = \frac{n(n-1)(n+1)}{6} = \frac{n(n^2-1)}{6}$$

Answer: The algorithm returns $\frac{n(n^2-1)}{6}$.

Part 2: Time Complexity The algorithm has three nested loops:

- Outer loop: k from 1 to $n-1$ (runs $n-1$ times)
- Middle loop: l from $k+1$ to n (runs up to $n-1$ times for each k)
- Inner loop: m from 1 to l (runs up to n times for each l)

The total number of operations is exactly what we calculated above: $\frac{n(n^2-1)}{6}$.
Since $\frac{n(n^2-1)}{6} = \frac{n^3-n}{6} = O(n^3)$:
Answer: The worst-case running time is $O(n^3)$.

10.3 Questions

1. **Network Flow Applications:** How could the Ford-Fulkerson algorithm be applied to solve real-world problems such as internet traffic routing or supply chain optimization? What modifications might be needed for practical implementations?
2. **Algorithm Complexity:** Compare the time complexity of the analyzed algorithm $O(n^3)$ with common sorting algorithms. In what scenarios might a cubic-time algorithm still be acceptable, and what optimization strategies could potentially reduce the complexity?

11 Final Reflections

This report provided a deep dive into foundational concepts in automata theory, computational models, and algorithmic analysis over the course of twelve weeks. From the basics of finite automata and regular languages to the complexities of Turing machines and network flow algorithms, the material emphasizes both theoretical depth and real-world relevance.

Key takeaways include:

- Understanding the equivalence of DFAs and NFAs and mastering state-based modeling.
- Applying algorithmic transformations such as minimization, powerset construction, and regular expression conversion.
- Recognizing the expressive power of Turing machines and their role in classifying decidable and undecidable problems.
- Employing Boolean logic and SAT encoding to solve combinatorial constraints, including Sudoku.
- Analyzing algorithm efficiency using Big O notation and applying techniques such as Ford-Fulkerson to optimize resource flow.

  Together, these topics form a cornerstone for further study in computation theory, programming languages, software verification, and complexity analysis. As technology evolves, the principles discussed remain crucial for designing robust and efficient systems, reaffirming the lasting value of theoretical computer science.