1 Week1

1.1 Summary

This chapter dives into finite automata, a key concept in understanding regular languages. At their core, finite automata are simple machines that process inputs and move between a set number of states based on defined rules. There are two main types: deterministic finite automata (DFA), which have a single, predictable transition for each state and input, and non-deterministic finite automata (NFA), which can move to multiple states at once for the same input.

A real-world example explored in this chapter is an electronic money system, where automata help model transactions between a customer, store, and bank. The system ensures valid transactions and prevents fraud—such as reusing the same digital money file multiple times. The automaton governs five key events: paying, canceling, shipping, redeeming, and transferring money, with each entity following strict rules to ensure everything runs smoothly.

Another important concept covered is how automata handle errors and ignore irrelevant actions. By allowing self-loops for unimportant inputs, an automaton can avoid breaking when faced with unexpected behavior. The chapter also examines potential pitfalls, such as cases where a store ships goods before confirming payment, leading to irreversible mistakes.

To better understand interactions between different automata, the product automaton is introduced. This technique combines the states of two automata—representing the store and bank—into a larger system with 28 states. This allows for systematic error-checking, such as confirming whether goods can be shipped without actual payment.

The chapter wraps up by analyzing which states are reachable and identifying failure points, like trying to cancel a payment that has already been processed. Ultimately, it highlights the importance of carefully structuring automata to avoid logical flaws in real-world applications.

1.2 Homework

1.3 Accepted Words for Vending Machine Automaton

The vending machine automaton accepts words that sum up to 25 cents, using symbols {5, 10}. The set of accepted words consists of:

- **(5,5,5,5,5)**

- **(5,5,5,10)**
- **(5,5,10,5)**
- **(5,10,5,5)**
- **(10,5,5,5)**
- **(5,10,10)**
- **(10,5,10)**
- **(10,10,5)**

1.4 Regular Expression for Variable Names

The accepted words for variable names must start with a letter (`l`), followed by any combination of letters (`l`) and digits (`d`), and must end with a terminal symbol (`t`).

The corresponding regular expression is:

$$l(l+d)^*t \tag{1}$$

where:

- `l` represents letters ({a, b, c, ..., z})
- `d` represents digits ({0, 1, 2, ..., 9})
- `t` is a terminal symbol (such as ;)

1.5 Regular Expression for Turnstile Automaton

The turnstile automaton has two states: `locked` and `unlocked`. The transitions occur as follows:

- `pay` moves from `locked` to `unlocked`.
- `push` moves from `unlocked` to `locked`.
- Multiple `pay` actions while unlocked keep it open.

The regular expression describing the accepted words is:

$$(pay(pay)*)?push \tag{2}$$

1.6 DFA Homework Solutions

Given the DFA defined by the following:

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

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

- Initial state: q_0
- Accepting state: $F = \{q_1\}$

1.7 Discussion Question

What are the practical and theoretical limits of automata complexity? At what point does an automaton become too complex to be useful?

2 Week2

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

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

Exercise 2 (Implementing DFA runs)

In `dfa.py`, implement the `run` method as follows:

```
class DFA:
    def __init__(self, Q, Sigma, delta, q0, F):
        self.Q = Q
        self.Sigma = Sigma
        self.delta = delta
        self.q0 = q0
        self.F = F

    def run(self, w):
        curr = self.q0
        for sym in w:
            if sym not in self.Sigma:
                raise ValueError(f"Symbol {sym!r} not in alphabet ")
            key = (curr, sym)
            if key in self.delta:
                curr = self.delta[key]
            elif curr in self.delta and sym in self.delta[curr]:
                curr = self.delta[curr][sym]
            else:
                return False
        return curr in self.F
```

Exercise 3 (Designing DFAs)

For each language over $\{a, b\}$, we give a 5-tuple definition and a schematic.

[a)]Words ending in "`ab`".
$Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, q_0, $F = \{q_2\}$.
Transitions: $\delta(q_0, a) = q_1$, $\delta(q_0, b) = q_0$; $\delta(q_1, a) = q_1$, $\delta(q_1, b) = q_2$; $\delta(q_2, a) = q_1$, $\delta(q_2, b) = q_0$. Words containing "`aba`" as a substring.
$Q = \{p_0, p_1, p_2, p_3\}$, $q_0 = p_0$, $F = \{p_3\}$.
Progress states track prefix matches of "aba"; on match move to p_3 and stay. Words with odd number of a's and odd number of b's.
$Q = \{(e, e), (e, o), (o, e), (o, o)\}$, start (e, e), final (o, o). Words with even number of a's and odd number of b's.
Same Q, start (e, e), final (e, o). Words in which any three consecutive characters contain at least one a.
$Q = \{r_0, r_1, r_2, r_3\}$, where r_k means last k symbols were b; start r_0, sink r_3, finals $\{r_0, r_1, r_2\}$. Words containing "`bbb`" as a substring.
Same Q, with final r_3 instead.

Exercise 4 (Complement Automaton)

Given $A = (Q, \Sigma, \delta, q_0, F)$, its complement A_0 is

$$A_0 = (Q, \Sigma, \delta, q_0, Q \setminus F).$$

In `dfa.py`:

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

Exercise 5 (Intersection Automaton)

Given A and B, the intersection C has:

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

Exercise 2.2.4 (ITALC)

[a)]Ends in "00": states s_0, s_1, s_2, $F = \{s_2\}$. Has three consecutive 0's: states t_0, t_1, t_2, t_3, $F = \{t_3\}$. Has substring "011": states u_0, u_1, u_2, u_3, $F = \{u_3\}$.

Summary

This week reinforced core concepts in automata theory and their practical relevance in computer science:

- **Finite-State Modeling**: We deepened our understanding of DFAs as abstract machines that precisely capture patterns in strings through state transitions, laying the groundwork for modeling sequential logic and protocol behaviors.
- **Regular Languages and Closure**: By designing DFAs for endings, substrings, parity checks, and forbidden patterns, we explored the expressive power of regular languages and saw how operations like complement and intersection enable algebraic manipulation of those languages.
- **Algorithmic Construction**: Implementing simulation (`run`), complementation (`refuse`), and intersection (product construction) methods highlighted how formal definitions translate into concrete algorithms—an essential skill in compiler design and text-processing tools.

- **Software Engineering Integration**: Bridging the theory with Python code reinforced good practices for representing transition functions, error handling for unexpected inputs, and writing clear, maintainable simulation routines.

Questions for Reflection

1. How might the complement and intersection constructions for DFAs be used in software testing for validating input constraints?
2. In what ways do DFA designs correspond to components in a compiler's lexical analyzer when tokenizing source code?
3. How can the principles of state minimization and transition completeness improve the efficiency of regular-expression engines in practical applications?
4. Can you think of another area in computer science—such as network protocol design or digital circuit verification—where DFA constructions play a crucial role?
5. How would you extend these DFA techniques to model nondeterministic or infinite-state behaviors encountered in real-world systems?

3 Week3

Homework 1 (Extended Transition Function)

[1.]**Describe** $L(\mathcal{A}^{(2)})$**.**
From the diagram, $\mathcal{A}^{(2)}$ accepts exactly those words of odd length in which every letter in an odd position is `a`. In set notation:

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

Compute $\hat{\delta}^{(1)}(1,\, abaa)$ **and** $\hat{\delta}^{(2)}(1,\, abba)$**.** Write out every step of the extended transition function.
(a) For $\mathcal{A}^{(1)}$, we have the transitions:

$$1 \xrightarrow{a} 2, \quad 2 \xrightarrow{b} 3, \quad 3 \xrightarrow{a} 2, \quad 2 \xrightarrow{a} 2.$$

Hence:

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

So $\hat{\delta}^{(1)}(1, abaa) = 2$.
(b) For $\mathcal{A}^{(2)}$, with

$$1 \xrightarrow{a} 2,\ 1 \xrightarrow{b} 3,\ 2 \xrightarrow{b} 1,\ 2 \xrightarrow{a} 1,\ 3 \xrightarrow{a,b} 3,$$

we compute:

$$\begin{aligned}\hat{\delta}^{(2)}(1,\,\varepsilon) &= 1,\\ \hat{\delta}^{(2)}(1,\,a) &= \delta^{(2)}(1,a) = 2,\\ \hat{\delta}^{(2)}(1,\,ab) &= \delta^{(2)}(2,b) = 1,\\ \hat{\delta}^{(2)}(1,\,abb) &= \delta^{(2)}(1,b) = 3,\\ \hat{\delta}^{(2)}(1,\,abba) &= \delta^{(2)}(3,a) = 3.\end{aligned}$$

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

Homework 2 (Product Automaton)

Let $\mathcal{A}^{(1)} = (Q_1, \Sigma, \delta^{(1)}, q_0^{(1)}, F^{(1)})$ and $\mathcal{A}^{(2)} = (Q_2, \Sigma, \delta^{(2)}, q_0^{(2)}, F^{(2)})$ with the given diagrams.

[1.]**Intersection Automaton $\mathcal{A}$:** Define

$$Q = Q_1 \times Q_2, \quad q_0 = (q_0^{(1)}, q_0^{(2)}), \quad F = F^{(1)} \times F^{(2)},$$

and for each $(p, q) \in Q$, $a \in \Sigma$:

$$\delta((p,q),a) = (\delta^{(1)}(p,a),\ \delta^{(2)}(q,a)).$$

In TikZ, the reachable portion of the product is:

1. [-¿,¿=stealth',shorten ¿=1pt,auto,node distance=2cm] [state,initial] (11) $(1,1)$; [state,accepting] (22)[right of=11] $(2,2)$; [state] (13)[below of=11] $(1,3)$; [state] (23)[right of=13] $(2,3)$; (11) edge node[above] a (22) (11) edge node[left] b (13) (22) edge node[below] b (23) (23) edge[loop right] node a, b (23);

2. **Proof of Intersection:** A word w is accepted by $\mathcal{A}$ iff its run ends in $(p, q) \in F^{(1)} \times F^{(2)}$, i.e. $p \in F^{(1)}$ and $q \in F^{(2)}$. By definition of δ, this holds exactly when w is accepted by both $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$. Hence $L(\mathcal{A}) = L(\mathcal{A}^{(1)}) \cap L(\mathcal{A}^{(2)})$.

3. **Union Automaton $\mathcal{A}'$:** To get $L(\mathcal{A}') = L(\mathcal{A}^{(1)}) \cup L(\mathcal{A}^{(2)})$, keep the same product transitions but replace

$$F' = (F^{(1)} \times Q_2) \ \cup \ (Q_1 \times F^{(2)}).$$

That is, accept any pair where at least one component is accepting.

Exercise 2.2.7 (Induction on $\hat{\delta}$)

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA and $q \in Q$ satisfy $\delta(q, a) = q$ for all $a \in \Sigma$. We prove by induction on $|w|$ that

$$\hat{\delta}(q, w) = q \quad \forall w \in \Sigma^*.$$

Base case: $|w| = 0$, so $w = \varepsilon$. By definition, $\hat{\delta}(q, \varepsilon) = q$.
Inductive step: Assume for some $n \geq 0$ the claim holds for all $|w| = n$. Let $|w| = n + 1$, write $w = xa$ with $|x| = n$ and $a \in \Sigma$. Then

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

where the second equality uses the IH and the third uses the loop hypothesis. Thus the statement holds for all n.

Reading and Discussion

Read ITALC Section 2.3 on *Nondeterministic Finite Automata*. Post an interesting question about NFAs on Discord and include it here:

"**Question:** How does the subset-construction algorithm ensure that an NFA's nondeterministic branching is faithfully simulated by a DFA without exponential blow-up in the recognized language?"

Summary

This week we extended our toolkit in automata theory:

- **Extended Transition Function:** Learned to track multi-step transitions $\hat{\delta}(q, w)$ by induction, underpinning formal proofs.
- **Product Automata:** Saw how intersection and union of regular languages correspond to Cartesian-product constructions on state sets and simple adjustments of final states.

- **Proof Techniques:** Practiced structural induction on string length, a key proof method throughout theoretical computer science.
- **Nondeterminism:** Began exploring NFAs and the subset construction, which bridges nondeterministic models with deterministic algorithms used in lexical analysis and model checking.

Questions

Reflect on how these topics extend to broader computer-science contexts:

1. How could structural induction on $\hat{\delta}$ be adapted to prove properties of pushdown automata or Turing machines?
2. In complex software systems (e.g., compilers, protocol verifiers), how might intersection and union automaton constructions enforce multiple constraints or specifications simultaneously?
3. What are the practical trade-offs of converting an NFA to a DFA via subset construction in real-world pattern-matching applications, considering state-space explosion and performance?

4 Week4

Homework 1 (Viewing a DFA as an NFA)

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We can regard A as an NFA by allowing its transition function to return singleton sets:

$$\delta_N(q, a) = \{\delta(q, a)\}, \quad \text{for each } q \in Q,\ a \in \Sigma.$$

[1.]For the concrete DFA below, simply reinterpret each arrow $p \xrightarrow{a} r$ as $p \xrightarrow{a} \{r\}$ in the NFA. In general, define $A' = (Q, \Sigma, \delta', q_0, F)$ with $\delta' : Q \times \Sigma \to P(Q)$ by $\delta'(q, a) = \{\delta(q, a)\}$. Clearly $L(A') = L(A)$ since the NFA has no additional nondeterminism. This construction satisfies $L(A') = L(A)$ by definition: each NFA branch follows the unique DFA transition.

Homework 2 (Power-Set Construction)

Let $N = (Q, \Sigma, \delta_N, q_0, F)$ be the given NFA. Its determinization is $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ where:

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

Example Determinization

Starting from $\{q_0\}$, we compute reachable subsets and draw the DFA:

[-¿,¿=stealth',shorten ¿=1pt,auto,node distance=2cm] [state,initial] (0) {q0}; [*state*](01)[*rightof* = 0]{q_0, q_1}; [*state*, *accepting*](1)[*belowof* = 01]{q_1, q_2}; [*state*](2)[*belowof* = 0]∅; (0)*edgenode*a(01)*edgenode*[*left*]b(0)(01)*edgenode*a(01)*edgenode*b(1)(1)*edge*[*loopright*]*node*a,b()(2)*edg*

Reading and Discussion

Read ITALC Sections 3.1–3.2.2 on determinization and the power-set automaton. Post an interesting question on Discord and include it here.

Summary

This week's lessons highlighted:

- **Equivalence of NFAs and DFAs**: Although NFAs allow multiple next states, they recognize exactly the same class of regular languages.
- **Canonical DFA Representation**: Every DFA is trivially an NFA by viewing transitions as singleton sets, reinforcing the shared model.
- **Power-Set Construction**: The subset algorithm systematically converts any NFA into an equivalent DFA, with states representing sets of NFA states and accepting those that contain an NFA final state.
- **Practical Considerations**: Determinization may introduce exponentially many states; in real engines, unreachable subsets are omitted and trap states help manage missing transitions.

Questions

1. In designing a lexical analyzer for a programming language, how would you apply the power-set construction to combine multiple token -recognition NFAs into a single efficient DFA?
2. What strategies can mitigate the state-explosion problem when determinizing large NFAs derived from complex regular expressions?
3. How does treating a DFA as an NFA simplify proofs of closure properties of regular languages under operations like concatenation and Kleene star?

5 Week5

Homework 1: Testing Equivalence and Minimization

Exercise 4.4.1 (Fig. 4.14)

The DFA has states $\{A, B, C, D, E, F, G, H\}$ with start A and final D. The transition table:

	0	1
$\to A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

Table-filling for distinguishability. Initially mark pairs where exactly one is final:

$$\{A, B\}, \{B, D\}, \{C, D\}, \{D, E\}, \{D, F\}, \{D, G\}, \{D, H\}, \{D, A\}, \ldots$$

Iterating on transitions, the only pairs that remain unmarked (i.e. equivalent) are:

$$(A, G),\ (B, F),\ (C, E).$$

Thus the equivalence classes are:

$$\{A, G\}, \{B, F\}, \{C, E\}, \{D\}, \{H\}.$$

Minimal DFA. Merging each equivalent class:

- $X = \{A, G\}$ (start),
- $Y = \{B, F\}$,
- $Z = \{C, E\}$,
- $W = \{D\}$ (final),
- $H' = \{H\}$.

Transitions on 0 and 1:

	0	1
$\to X$	Y	X
Y	X	Z
Z	W	Y
W^*	W	X
H'	X	W

Exercise 4.4.2 (Fig. 4.15)

The nine-state DFA (start A, finals C, I) has no equivalent pairs under the table-filling algorithm, so it is already minimal.

Homework 2: Regular-Expression Derivation

Exercise 3.2.1 (First DFA)

States q_1, q_2, q_3, start q_1, final q_3; table:

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

R^0_{ij} (direct transitions):

$$R^0 = \begin{pmatrix} 1 & 0 & \emptyset \\ 1 & \emptyset & 0 \\ \emptyset & 1 & 0 \end{pmatrix}$$

(rows $i = 1, 2, 3$; cols $j = 1, 2, 3$.)

R^1_{ij} (allowing paths via q_1):

$$R^1 = \begin{pmatrix} 1(1)^* & 0 \cup 1(1)^*0 & \emptyset \\ 1(1)^* & 1(1)^*0 & 0 \\ \emptyset & 1 & 0 \end{pmatrix}$$

R^2_{ij} (via q_2): Omitting intermediate algebra, one finds:

$$R^2_{13} = (0 \cup 1(1)^*0)(1(1)^*0)^*0,$$

and in particular all routes from q_1 to q_3 require two zeros with only zeros thereafter.

Final regex: Strings ending in at least two `0`'s:

$$L = (0 \cup 1)^* \, 00 \, 0^*.$$

Exercise 3.2.2 (Second DFA)

Table:

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

By repeated elimination one shows that L is the set of strings with an odd number of 1's (zeros can appear arbitrarily). A concise regex is:

$$L = 0^*(10^*10^*)^*10^*.$$

Reading and Discussion

Question posted on Discord: "How does minimization depth (number of table-filling rounds) relate empirically to language complexity in real-world regex engines?"

Summary

This week's material deepened both our theoretical understanding and practical skills around the core equivalence relationship between finite automata and regular expressions. We began by rigorously defining when two states of a DFA are équivalent

1. In large-scale pattern-matching engines, how might one decide whether to use a minimized DFA or a directly-eliminated-regex for efficiency?
2. Can the table-filling algorithm be parallelized to speed up minimization on very large automata, and what are the trade-offs?
3. How do the properties of equivalent-state merging inform optimizations in hardware finite-state machines (e.g., in FPGA design)?

6 Week 67

6.1 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

$$L = \{10^n10^m10^{m+n} : n, m \in N\}$$

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 $\{0^n1^n\}$), 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. For instance:

- $L_1 = \{M \mid M \text{ halts on itself}\}$ is neither decidable nor RE.
- $L_2 = \{(M, w) \mid M \text{ halts on } w\}$ is RE but not decidable.
- $L_3 = \{(M, w, k) \mid M \text{ halts on } w \text{ in at most } k \text{ steps}\}$ is decidable.

Finally, *Turing completeness* is the property of a system that can simulate any TM. Questions about the limits of computation—such as whether all RE languages are recursive—drive the study of computation theory.

6.2 Homework

1. TM for $L = \{10^n : n \in N\}$, return 10^{n+1}

This Turing machine moves to the end of the 0s, adds another 0, and halts.

State	1	0	B
q0	q1,1,R		
q1		q1,0,R	q2,0,L
q2			q3,1,L
q3		q3,0,L	q4,1,R
q4		q4,0,R	HALT

Example: Input: 1000 ⇒ Output: 10000

2. TM for $L = \{10^n\}$, return 1

This machine replaces all symbols and writes a single 1.

```
State  | 1       | 0       | B
-------------------------------
q0     | q1,X,R  | q0,X,R  | q2,1,R
q1     |         |         | HALT
```

3. TM that swaps 0s and 1s in any binary string

```
State  | 0       | 1       | B
-------------------------------
q0     | q0,1,R  | q0,0,R  | HALT
```

Exercise 1: Decide whether the languages are decidable, r.e., or co-r.e.

1. $L_1 = \{M \mid M \text{ halts on itself}\}$: **Not decidable**, **not r.e.**
2. $L_2 = \{(M, w) \mid M \text{ halts on } w\}$: **r.e.**, **not decidable**
3. $L_3 = \{(M, w, k) \mid M \text{ halts on } w \text{ in at most } k \text{ steps}\}$: **Decidable**

Exercise 2: Determine the truth of each statement

1. **True**: The union of two decidable languages is decidable.
2. **True**: Decidable languages are closed under complement.
3. **True**: Decidable languages are closed under Kleene star.
4. **True**: r.e. languages are closed under union.
5. **False**: r.e. languages are not closed under complement.
6. **True**: r.e. languages are closed under Kleene star.

6.3 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 = $\{\prime \backslash \infty \backslash \neg \backslash \in N\}, and how do Turing machines overcome this limitation to p$

7 Week 8/9

7.1 Summary

$\mathcal{B}\rangle\}\mathcal{O}\backslash\wr\sqcup\dashv\sqcup\rangle\wr\backslash\Leftrightarrow\dashv\updownarrow\int\wr\|\backslash\wr\sqsupseteq\backslash\dashv\int\mathcal{L}\dashv\backslash\lceil\dashv\sqcap\backslash\wr\sqcup\dashv\sqcup\rangle\wr\backslash\Leftrightarrow\rangle\int\dashv\Updownarrow\dashv\sqcup\langle\rceil\Updownarrow\dashv\sqcup\rangle\rfloor\dashv\updownarrow\sqcup\wr\updownarrow\sqcap\int\rceil\lceil\sqcup\wr\lceil\nwarrow$
$\int\rfloor\nabla\rangle\lfloor\rceil\sqcup\langle\rceil\sqcap\sqrt{\sqrt{}}\rceil\nabla\lfloor\wr\sqcap\backslash\lceil\wr\{\dashv\{\sqcap\backslash\rfloor\sqcup\rangle\wr\backslash$'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) $\in \mathcal{O}(g(n))means that for large enough n, f(n) \leq M \cdot g(n). The properties of Big O, such as closure under addition, scaling by constants, and asymptotic comparis$

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.

7.2 Homework

Exercise 1: Order of Growth (Slow to Fast)

Order the functions by their asymptotic growth:

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

Exercise 2: Basic Big O Properties

Let $f, g, h : N \to R_{\geq 0}$.

1. $f \in \mathcal{O}(f)$
2. If $c > 0$, then $\mathcal{O}(c \cdot f) = \mathcal{O}(f)$
3. If $f \leq g$ for large inputs, then $\mathcal{O}(f) \subseteq \mathcal{O}(g)$
4. If $\mathcal{O}(f) \subseteq \mathcal{O}(g)$, then $\mathcal{O}(f + h) \subseteq \mathcal{O}(g + h)$
5. If $h(n) > 0$ and $\mathcal{O}(f) \subseteq \mathcal{O}(g)$, then $\mathcal{O}(f \cdot h) \subseteq \mathcal{O}(g \cdot h)$

Exercise 3: Useful Asymptotic Bounds

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

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

Exercise 4: Growth Function Comparisons

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

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 $O(n^2)$

7.3 Questions

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? 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?