notes

Master GATE CS Notes: Comprehensive Reference Guide for Question Banks (Volume 1 & 2)

📚 COMPREHENSIVE INDEX

Volume 1: Engineering Mathematics, Discrete Mathematics, and General Aptitude

1. Discrete Mathematics: Set Theory & Algebra (171 Questions)

2. Discrete Mathematics: Mathematical Logic (77 Questions)

3. Discrete Mathematics: Graph Theory (83 Questions)

4. Discrete Mathematics: Combinatory (50 Questions)

5. Engineering Mathematics: Calculus (150 Questions)

6. Engineering Mathematics: Linear Algebra

7. Engineering Mathematics: Probability and Statistics

8. General Aptitude

Volume 2: Core Computer Science Subjects

1. Theory of Computation

- 1.2 Regular Languages
- 1.3 Context-Free Languages
- 1.4 Turing Machines
- 1.5 Computability
- 1.6 Complexity Theory

2. Algorithms and Data Structures

- 2.1 Algorithm Analysis
- 2.2 Sorting and Searching
- 2.3 Graph Algorithms
- 2.4 Dynamic Programming
- 2.5 Data Structures

3. Computer Organization and Architecture

- 3.1 Number Systems
- 3.2 Boolean Algebra
- 3.3 Processor Design
- 3.4 Memory Hierarchy
- 3.5 I/O Systems
- 3.6 Ethernet Bridging

4. Operating Systems

- 4.1 Process Management
- 4.2 Memory Management
- 4.3 File Systems
- 4.4 Synchronization
- 4.5 CPU Scheduling

5. Database Management Systems

- 5.1 ER Model
- 5.2 Relational Model
- 5.3 SQL
- 5.4 Normalization
- 5.5 Transactions
- 5.6 File Organization

6. Computer Networks

- 6.1 Network Models
- 6.2 Physical Layer

7. Software Engineering

8. Compiler Design

🔍 Quick Navigation Guide

- **Search**: Use Ctrl+F to find specific topics instantly
- **Formulas**: Look for $ delimiters around mathematical expressions
- **GATE Tips**: Search for "GATE trap" and "Common pattern" annotations
- **Cross-references**: Follow "(See X.Y)" links to related sections
- **Examples**: Each section includes worked problems and solutions

---

This master note is a compiled, exhaustive reference designed specifically for solving problems from the provided GATE question banks (Volume 1: Engineering Math, Discrete Math, General Aptitude; Volume 2: Core CS Subjects). It covers **every topic and subtopic** from the Tables of Contents in both volumes, drawing from standard GATE syllabus knowledge, key formulas, theorems, proofs, algorithms, examples, and common problem-solving patterns. The structure mirrors the TOCs for easy navigation. Each subtopic includes:

- **Key Concepts**: Core definitions and ideas.

- **Formulas/Theorems**: Essential equations, rules, and proofs (with brief derivations where closed-ended math is involved).
- **Problem-Solving Tips**: Strategies for typical GATE questions, with recall triggers.
- **Examples**: Quick illustrative problems (inspired by bank patterns).

Use this as a "one-stop recall sheet"—scan for keywords from a problem, review formulas, and apply tips. Total coverage: ~100% of listed subtopics (1,700+ questions across volumes). Organized by volume for clarity.

How to Use These Notes:

- **Quick Lookup**: Use Ctrl+F to find specific topics
- **Formula Reference**: All key formulas are marked with $ delimiters
- **Problem Patterns**: Look for "GATE trap" and "Common pattern" annotations
- **Cross-references**: "(See X.Y)" links to related topics
- **Practice Strategy**: Cover examples first, then attempt similar problems

---

Volume 1: Engineering Mathematics, Discrete Mathematics, and General Aptitude

1. Discrete Mathematics: Combinatory (50 Questions)

1.1 Balls In Bins (4) {#11-balls-in-bins-4}

Key Concepts: Models distributing indistinguishable/distinguishable objects into bins; stars and bars for indistinguishable cases. Core decision tree: (1) Are balls distinguishable? (2) Are bins distinguishable? (3) Can bins be empty?

Formulas/Theorems:

- **Indistinguishable balls, distinct bins, bins can be empty**: $\binom{n+k-1}{k-1}$ (stars and bars; proof: arrange n stars and k-1 bars in a line)
- **Indistinguishable balls, distinct bins, no empty bin**: $\binom{n-1}{k-1}$ (place one in each bin first)
- **Distinct balls, distinct bins**: k^n (each ball has k choices)
- **Distinct balls, distinct bins, no empty bin**: $k! \cdot S(n,k)$ where $S(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}(k-i)^n$ (Stirling second kind)
- **Distinct balls, indistinct bins**: $\sum_{i=1}^{\min(n,k)} S(n,i)$ (Bell numbers)

Problem-Solving Tips:

- GATE trap: Forgetting to check if bins/balls are labeled
- Common pattern: "distribute among groups" → indistinct bins
- Quick check: For n=k=2, should get specific known values

- Inclusion-exclusion for "at most r per bin": subtract bad arrangements

Examples:

- 5 identical balls, 3 bins: $\binom{7}{2} = 21$
- 3 distinct balls, 2 bins (no empty): $2^3 - 2 = 6$
- 4 distinct balls, 3 indistinct bins: $S(4,1) + S(4,2) + S(4,3) = 1 + 7 + 6 = 14$
- Distribute 10 identical items, each person gets at least 2, among 3 people: $\binom{10-6+3-1}{3-1} = \binom{6}{2} = 15$

1.2 Combinatory (22) {#12-combinatory-22}

Key Concepts: Combinatorics studies finite or countable discrete structures. Core principle: systematic enumeration without explicit listing.

Fundamental Counting Principles:

1. Addition Principle (Disjoint Union):

If task A can be done in m ways and task B in n ways, and A and B are mutually exclusive, then "A OR B" can be done in m+n ways.

- Formal: If $A \cap B = \emptyset$, then $|A \cup B| = |A| + |B|$
- Extension: $|A_1 \cup A_2 \cup \cdots \cup A_k| = \sum |A_i|$ when pairwise disjoint

2. Multiplication Principle (Cartesian Product):

If task A can be done in m ways, and for each way, task B can be done in n ways, then "A AND B" can be done in m×n ways.

- Formal: $|A \times B| = |A| \cdot |B|$
- Extension: For k independent tasks with n_i choices each: $\prod_{i=1}^{k} n_i$

Permutations: Ordered arrangements

Linear Permutations of n Distinct Objects:

- Full permutation: $P(n,n) = n!$
- Derivation: n choices for first position, (n-1) for second, ..., 1 for last
- Product: $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n!$

r-Permutations (arrangements of r from n):

- Formula: $P(n,r) = \frac{n!}{(n-r)!}$
- Derivation: n choices for first, (n-1) for second, ..., (n-r+1) for r-th position
- Product: $n(n-1)(n-2)\cdots(n-r+1) = \frac{n!}{(n-r)!}$
- Alternative notation: ${}_nP_r$ or A_n^r

Permutations with Repetitions (Multiset Permutations):

- Given n objects where n_1 are type 1, n_2 are type 2, ..., n_k are type k
- Where $n_1 + n_2 + \cdots + n_k = n$
- Formula: $\frac{n!}{n_1!n_2!\cdots n_k!}$
- **Proof**: If all objects were distinct: n! arrangements. But objects of same type are indistinguishable. Dividing by n_i! for each type removes overcounting from permuting identical items.
- Example: "SUCCESS" has 7 letters: S(3), U(1), C(2), E(1)
  - Arrangements: $\frac{7!}{3!\cdot1!\cdot2!\cdot1!} = \frac{5040}{12} = 420$

Circular Permutations:

- Arranging n distinct objects in a circle: $(n-1)!$
- **Derivation**: Fix one object to break rotational symmetry (prevents counting rotations as different). Arrange remaining (n-1) objects linearly.
- If clockwise/counterclockwise same (e.g., necklace): $\frac{(n-1)!}{2}$
- With r identical items: More complex, use Burnside's lemma

Derangements (!n): Permutations where no element appears in its original position

Formula: $D_n = n!\sum_{i=0}^{n}\frac{(-1)^i}{i!}$

Alternative: $D_n = n!\left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + \frac{(-1)^n}{n!}\right)$

Recurrence: $D_n = (n-1)[D_{n-1} + D_{n-2}]$ for $n \geq 2$, with $D_0 = 1, D_1 = 0$

Derivation via Inclusion-Exclusion:

Let A_i = permutations where element i is in position i

- Want: $|\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}|$ (none in original position)
- By inclusion-exclusion: $D_n = n! - |A_1 \cup A_2 \cup \cdots \cup A_n|$
- $|A_i| = (n-1)!$ (fix element i, permute rest)
- $|A_i \cap A_j| = (n-2)!$ (fix i and j)
- $|A_{i_1} \cap \cdots \cap A_{i_k}| = (n-k)!$
- Number of k-intersections: $\binom{n}{k}$
- Therefore: $D_n = n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \cdots + (-1)^n\binom{n}{n}0!$
- Simplify: $D_n = n!\sum_{k=0}^{n}\frac{(-1)^k}{k!}$

Asymptotic: $D_n \approx \frac{n!}{e}$ (since $e = \sum_{k=0}^{\infty}\frac{1}{k!}$)

Probability: Fraction of derangements = $\frac{D_n}{n!} \approx \frac{1}{e} \approx 0.368$ (independent of n for large n!)

Small values:

- $D_0 = 1$ (empty permutation)
- $D_1 = 0$ (impossible to derange 1 element)
- $D_2 = 1$ (swap)
- $D_3 = 2$ (231, 312)
- $D_4 = 9$
- $D_5 = 44$

Applications:

- Hat-check problem: n people check hats, retrieve randomly. Probability nobody gets own hat ≈ 1/e
- Secret Santa arrangements where nobody draws self
- Rencontres problem

Combinations: Unordered selections

r-Combinations from n objects:

- Formula: $C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$
- Derivation: $P(n, r)$ counts ordered selections. Each unordered set of r elements corresponds to r! orderings.
- Thus: $C(n, r) = \frac{P(n,r)}{r!} = \frac{n!}{r!(n-r)!}$

Problem-Solving Tips:

- Decide: Does order matter? (Permutation) or not? (Combination)
- For restrictions: Use complementary counting or case analysis
- Symmetry: Look for ways to reduce by symmetry
- GATE trap: Distinguishing between "at least one" vs "exactly one"
- Overcounting identical items: Divide by appropriate factorials
- Use generating functions for complex constraints

Examples:

1. Arrange "MISSISSIPPI" (11 letters: M(1), I(4), S(4), P(2)):
   - $\frac{11!}{1!\cdot4!\cdot4!\cdot2!} = \frac{39916800}{1\cdot24\cdot24\cdot2} = \frac{39916800}{1152} = 34650$
2. Derangements of 4 items {1,2,3,4}:
   - $D_4 = 4! \sum_{i=0}^{4} \frac{(-1)^i}{i!} = 24(1 - 1 + \frac{1}{2} - \frac{1}{6} + \frac{1}{24}) = 24 \cdot \frac{3}{8} = 9$
   - Explicit: 2143, 2341, 2413, 3142, 3412, 3421, 4123, 4312, 4321
3. Arrange 5 people in a circle: $(5 - 1)! = 24$
4. Words from "COMPUTER" with vowels together:
   - Treat OUE as one unit: 6 units (CMPTR + OUE)
   - Arrange 6 units: $6!$
   - Arrange 3 vowels within: $3!$

- Total: $6! \cdot 3! = 720 \cdot 6 = 4320$

1.3 Counting (4) {#13-counting-4}

Key Concepts: Fundamental enumeration techniques form the basis of combinatorial analysis. Bijections establish equinumerosity between sets.

Bijective Proofs (Combinatorial Proofs):

- To prove $|A| = |B|$, construct bijection $f : A \to B$
- Elegant alternative to algebraic manipulation
- Example: $\binom{n}{k} = \binom{n}{n-k}$
  - Bijection: Each k-subset corresponds to its complement (n-k)-subset

Counting Functions:

1. Functions from A to B ($|A| = n, |B| = k$):

- Total functions: k^n (each of n elements has k choices)
- **Proof**: By multiplication principle

2. Injective Functions (One-to-one):

- Requires $n \leq k$ (pigeonhole principle)
- Count: $P(k, n) = \frac{k!}{(k-n)!}$
- **Derivation**: First element: k choices, second: k-1, ..., n-th: k-n+1
- $k(k-1)(k-2)\cdots(k-n+1) = \frac{k!}{(k-n)!}$

3. Surjective Functions (Onto):

- Requires $n \geq k$ (can't cover k elements with fewer than k)
- Count: Uses inclusion-exclusion with Stirling numbers
- Formula: $k! \cdot S(n, k)$ where $S(n, k)$ is Stirling number of second kind
- $S(n, k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n$

4. Bijective Functions:

- Requires $n = k$ (bijection exists only between equinumerous sets)
- Count: $n!$ (permutations of range)

Inclusion-Exclusion Principle (PIE):

Two Sets:

$|A \cup B| = |A| + |B| - |A \cap B|$

Three Sets:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

General Formula (n sets):

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_i |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n-1} |A_1 \cap \cdots \cap A_n|$$

Compact notation:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{\emptyset \neq S \subseteq \{1,\ldots,n\}} (-1)^{|S|-1} \left|\bigcap_{i \in S} A_i\right|$$

Complement form (often easier):

$$\left|\overline{A_1 \cup \cdots \cup A_n}\right| = |U| - \sum_i |A_i| + \sum_{i<j} |A_i \cap A_j| - \cdots$$

Applications:

1. Derangements (as seen in 1.2):

Count permutations with no fixed point using PIE

2. Surjective function count:

- Map n distinct balls to k distinct bins with no empty bin
- A_i = functions missing bin i
- Want: $k^n - |A_1 \cup \cdots \cup A_k|$
- $|A_i| = (k-1)^n$ (avoid one bin)
- $|A_i \cap A_j| = (k-2)^n$ (avoid two bins)
- Result: $\sum_{j=0}^{k} (-1)^j \binom{k}{j} (k-j)^n$

3. Euler's Totient Function:

$\phi(n)$ counts integers in $\{1, \ldots, n\}$ coprime to n

- Use PIE over prime divisors of n
- If $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$:
- $\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right)$

Double Counting (Counting in two ways):

- Count same set using two different methods
- Equate results to prove identity
- Example: $\sum_{k=0}^{n} \binom{n}{k} = 2^n$
  - LHS: Count k-subsets for each k
  - RHS: Each element in/out → 2^n subsets

Problem-Solving Strategy:

1. Identify what to count (be precise about "distinct" vs "identical")

2. Check if direct formula applies (permutation/combination)
3. If restrictions exist, consider:
   - Complementary counting
   - Inclusion-exclusion
   - Case analysis
   - Bijection to simpler problem
4. For existence questions: Pigeonhole principle

Examples:

1. Ways to choose 2 from 5 **with order**: $P(5,2) = \frac{5!}{3!} = 20$
2. Ways to choose 2 from 5 **without order**: $\binom{5}{2} = 10$
3. Surjective functions from $\{1,2,3\}$ to $\{A,B\}$:
   - Total functions: $2^3 = 8$
   - Missing A: 1 (all to B)
   - Missing B: 1 (all to A)
   - Surjective: $8 - 2 = 6$
   - Or: $2! \cdot S(3,2) = 2 \cdot 3 = 6$
4. Integers ≤100 divisible by 2 or 3:
   - Div by 2: $\lfloor 100/2 \rfloor = 50$
   - Div by 3: $\lfloor 100/3 \rfloor = 33$
   - Div by 6: $\lfloor 100/6 \rfloor = 16$
   - By PIE: $50 + 33 - 16 = 67$

1.4 Generating Functions (6) {#14-generating-functions-6}

Key Concepts: Generating functions transform counting problems into algebraic manipulations of formal power series. They encode sequences as coefficients of polynomials or infinite series.

Philosophy: Instead of finding closed form directly, encode the sequence as a function, manipulate algebraically, then extract coefficients.

Ordinary Generating Functions (OGF):

For sequence $\{a_0, a_1, a_2, \ldots\}$, the OGF is:

$G(x) = a_0 + a_1 x + a_2 x^2 + \cdots = \sum_{n=0}^{\infty} a_n x^n$

Key OGFs:

1. **Geometric series**: $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots = \sum_{n=0}^{\infty} x^n$
   - Valid for $|x| < 1$ (but we treat as formal series)
2. **Binomial series**: $(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$
   - For negative/fractional n: $(1+x)^{-n} = \sum_{k=0}^{\infty} \binom{n+k-1}{k} (-1)^k x^k$

3. **Generalized geometric**: $\frac{1}{(1-x)^k} = \sum_{n=0}^{\infty} \binom{n+k-1}{k-1} x^n$
   - **Proof**: Differentiate $\frac{1}{1-x}$ repeatedly
   - Interpretation: Ways to place n indistinguishable balls in k bins
4. **Exponential**: $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

Operations on Generating Functions:

1. Addition: If $A(x) = \sum a_n x^n$ and $B(x) = \sum b_n x^n$:

- $A(x) + B(x) = \sum (a_n + b_n) x^n$
- Interpretation: Two independent ways to achieve outcome

2. Multiplication (Convolution):

- $A(x) \cdot B(x) = \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} a_k b_{n-k} \right) x^n$
- Coefficient of x^n: $c_n = \sum_{k=0}^{n} a_k b_{n-k}$
- Interpretation: Sequential choices or compositions

3. Scalar multiplication: $c \cdot A(x) = \sum (c \cdot a_n) x^n$

4. Differentiation: $A'(x) = \sum n \cdot a_n x^{n-1}$

- Brings down exponent as coefficient

5. Integration: $\int A(x) dx = \sum \frac{a_n}{n+1} x^{n+1}$

6. Substitution: Replace x with $f(x)$

Coefficient Extraction:

Notation: $[x^n]G(x)$ denotes coefficient of x^n in $G(x)$

Methods:

1. **Series expansion**: Expand and read coefficient
2. **Partial fractions**: Decompose rational functions
3. **Binomial theorem**: For powers of binomials
4. **Residue theorem** (advanced): Complex analysis

Applications:

1. Counting Compositions:

- Compositions of n (ordered partitions): How many ways to write $n = a_1 + a_2 + \cdots + a_k$ where $a_i \geq 1$?
- Each position: Choose positive integer
- GF for one part: $x + x^2 + x^3 + \cdots = \frac{x}{1-x}$
- For k parts: $\left(\frac{x}{1-x} \right)^k$

- For any number of parts: $\sum_{k=1}^{\infty} \left(\frac{x}{1-x}\right)^k = \frac{x/(1-x)}{1-x/(1-x)} = \frac{x}{1-2x}$
- Coefficient: $[x^n]\frac{x}{1-2x} = 2^{n-1}$ for $n \geq 1$

2. Integer Partitions:

- Partitions of n (unordered): $n = a_1 + a_2 + \cdots$ where $a_1 \geq a_2 \geq \cdots$
- GF (Euler): $P(x) = \prod_{i=1}^{\infty} \frac{1}{1-x^i}$
- Interpretation: For each i, choose how many i's to include: $1 + x^i + x^{2i} + \cdots = \frac{1}{1-x^i}$
- No closed form, but recurrences exist

3. Coin Change Problem:

- Coins of denominations $d_1, d_2, \ldots, d_k$, ways to make amount n?
- GF: $\prod_{i=1}^{k}(1 + x^{d_i} + x^{2d_i} + \cdots) = \prod_{i=1}^{k} \frac{1}{1-x^{d_i}}$
- Example: Coins {1,2,5}, coefficient of x^n in $\frac{1}{(1-x)(1-x^2)(1-x^5)}$

4. Recurrence Relations:

- Fibonacci: $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0, F_1 = 1$
- Let $G(x) = \sum_{n=0}^{\infty} F_n x^n$
- Multiply recurrence by x^n, sum over n: $\sum_{n=2}^{\infty} F_n x^n = \sum_{n=2}^{\infty} F_{n-1} x^n + \sum_{n=2}^{\infty} F_{n-2} x^n$
- $G(x) - F_0 - F_1 x = x(G(x) - F_0) + x^2 G(x)$
- $G(x) - x = xG(x) + x^2 G(x)$
- $G(x) = \frac{x}{1-x-x^2}$
- Partial fractions: $\frac{x}{1-x-x^2} = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi x} - \frac{1}{1-\psi x}\right)$ where $\phi = \frac{1+\sqrt{5}}{2}, \psi = \frac{1-\sqrt{5}}{2}$
- Extract: $F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}$ (Binet's formula)

Exponential Generating Functions (EGF):

For sequence $\{a_0, a_1, a_2, \ldots\}$:

$\hat{G}(x) = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!}$

When to use EGF: Problems involving **labeled** objects or **permutations**

Key property: Product of EGFs corresponds to labeled compositions

- If $\hat{A}(x) \cdot \hat{B}(x) = \hat{C}(x)$, then:
- $c_n = \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k}$
- The $\binom{n}{k}$ accounts for choosing which k objects get labels from first set

Example - Permutations with restrictions:

- Derangements: $D(x) = \sum_{n=0}^{\infty} D_n \frac{x^n}{n!}$
- Recurrence leads to: $D(x) = \frac{e^{-x}}{1-x}$

- Verifies: $D_n \approx \frac{n!}{e}$

Problem-Solving Strategy:

1. **Model the problem**: What choices are being made?
2. **Write GF for single choice**: Based on constraints
3. **Combine using operations**: Multiplication for sequential, addition for alternatives
4. **Simplify algebraically**: Use known series, partial fractions
5. **Extract coefficient**: Use appropriate technique

GATE Tips:

- OGF for unordered/unlabeled problems
- EGF for ordered/labeled problems
- Product of GFs → Convolution of sequences
- For rational GF $\frac{P(x)}{Q(x)}$: Roots of $Q(x)$ determine growth rate
- Partial fractions essential for extraction

Examples:

1. **Ways to make change for 5 rupees using 1,2 rupee coins**:
   - GF: $(1 + x + x^2 + \cdots)(1 + x^2 + x^4 + \cdots) = \frac{1}{(1-x)(1-x^2)}$
   - $= \frac{1}{(1-x)^2(1+x)}$
   - Partial fractions: $\frac{A}{1-x} + \frac{B}{(1-x)^2} + \frac{C}{1+x}$
   - Coefficient of x^5: 3 ways (5×1, 3×1+1×2, 1×1+2×2)
2. **Number of binary strings of length n with no consecutive 1s**:
   - Let a_n be count
   - Recurrence: $a_n = a_{n-1} + a_{n-2}$ (end in 0 or end in 01)
   - $a_0 = 1, a_1 = 2$
   - GF: $G(x) = \frac{1+x}{1-x-x^2}$ (Fibonacci-like)
3. **Partition of 5**:
   - $[x^5] \prod_{i=1}^{5} \frac{1}{1-x^i} = [x^5] \frac{1}{(1-x)(1-x^2)(1-x^3)(1-x^4)(1-x^5)}$
   - Computing: 7 partitions (5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1)

1.5 Modular Arithmetic (2) {#15-modular-arithmetic-2}

Key Concepts: Modular arithmetic is arithmetic "with wraparound" at a modulus. Foundation for number theory, cryptography, hashing, and algorithm design.

Congruence Relation:

Definition: $a \equiv b \pmod{m}$ if m divides (a-b)

- Notation: $m|(a-b)$ or $a - b = km$ for some integer k

- Equivalently: a and b have the same remainder when divided by m
- $a \bmod m$ denotes the remainder: unique r where $0 \leq r < m$ and $a \equiv r \pmod{m}$

Properties (Congruence is an equivalence relation):

1. **Reflexive**: $a \equiv a \pmod{m}$
2. **Symmetric**: If $a \equiv b \pmod{m}$, then $b \equiv a \pmod{m}$
3. **Transitive**: If $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$

Arithmetic Operations (Compatible with congruence):

If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then:

1. **Addition**: $a + c \equiv b + d \pmod{m}$
2. **Subtraction**: $a - c \equiv b - d \pmod{m}$
3. **Multiplication**: $ac \equiv bd \pmod{m}$
4. **Exponentiation**: $a^k \equiv b^k \pmod{m}$ for $k \geq 0$

Division (Careful!):

- $ac \equiv bc \pmod{m}$ does NOT imply $a \equiv b \pmod{m}$ in general
- **Cancellation law**: If $ac \equiv bc \pmod{m}$ and $\gcd(c, m) = 1$, then $a \equiv b \pmod{m}$
- **Proof**: $m | c(a - b)$. Since $\gcd(c, m) = 1$, by Euclid's lemma, $m | (a - b)$

Modular Inverse:

Definition: $a^{-1} \pmod{m}$ is value x where $ax \equiv 1 \pmod{m}$

Existence: a^{-1} exists mod m $\Leftrightarrow \gcd(a, m) = 1$ (a and m are coprime)

Finding inverse - Extended Euclidean Algorithm:

- Given $\gcd(a, m) = 1$, extended Euclid finds integers x, y where $ax + my = 1$
- Taking mod m: $ax \equiv 1 \pmod{m}$, so $x = a^{-1}$
- Time complexity: $O(\log \min(a, m))$

Example: Find $3^{-1} \pmod{7}$

- Extended Euclid on (7,3):
  - $7 = 2(3) + 1$
  - $1 = 7 - 2(3)$
  - So $1 \cdot 7 + (-2) \cdot 3 = 1$
  - Thus $(-2) \cdot 3 \equiv 1 \pmod{7}$
  - $-2 \equiv 5 \pmod{7}$
  - Answer: $3^{-1} \equiv 5 \pmod{7}$
- Verification: $3 \cdot 5 = 15 = 2 \cdot 7 + 1 \equiv 1 \pmod{7}$ ✓

Linear Congruences:

Problem: Solve $ax \equiv b \pmod{m}$

Solution:

- Let $d = \gcd(a, m)$
- **If** $d \nmid b$: No solution
- **If** $d|b$: d solutions exist
  - Divide through by d: $(a/d)x \equiv (b/d) \pmod{m/d}$
  - Now $\gcd(a/d, m/d) = 1$, so inverse exists
  - $x \equiv (b/d) \cdot (a/d)^{-1} \pmod{m/d}$
  - All solutions: $x \equiv x_0 + k(m/d)$ for $k = 0, 1, \ldots, d-1$

Chinese Remainder Theorem (CRT):

Problem: Solve system of congruences

$x \equiv a_1 \pmod{m_1}$

$x \equiv a_2 \pmod{m_2}$

$\vdots$

$x \equiv a_k \pmod{m_k}$

where m_i are pairwise coprime: $\gcd(m_i, m_j) = 1$ for $i \neq j$

Theorem: Unique solution modulo $M = m_1 m_2 \cdots m_k$

Construction:

1. Let $M_i = M/m_i$ (product of all except m_i)
2. Find $y_i = M_i^{-1} \pmod{m_i}$ (inverse exists since $\gcd(M_i, m_i) = 1$)
3. Solution: $x \equiv \sum_{i=1}^{k} a_i M_i y_i \pmod{M}$

Proof idea: M_i is divisible by all m_j except m_i, so $M_i y_i \equiv 0 \pmod{m_j}$ for $j \neq i$ and $M_i y_i \equiv 1 \pmod{m_i}$

Euler's Totient Function $\phi(n)$:

Definition: $\phi(n)$ = count of integers in $\{1, 2, \ldots, n\}$ coprime to n

Values:

- $\phi(1) = 1$
- $\phi(p) = p - 1$ for prime p
- $\phi(p^k) = p^k - p^{k-1} = p^{k-1}(p-1)$ (exclude multiples of p)
- **Multiplicative**: If $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m)\phi(n)$

Formula: For $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$:

$\phi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$

Derivation: By inclusion-exclusion on prime divisors (see section 1.3)

Properties:

1. $\sum_{d|n} \phi(d) = n$ (sum over divisors)
2. For n>2, $\phi(n)$ is even

Euler's Theorem:

Theorem: If $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$

Proof (Group theory):

- Consider multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$ of elements coprime to n
- This group has order $\phi(n)$
- By Lagrange's theorem, order of any element divides group order
- So $a^{\phi(n)} \equiv 1 \pmod{n}$

Consequence - Fast modular exponentiation:

- To compute $a^k \bmod n$ where k is large:
- Reduce exponent: $a^k = a^{k \bmod \phi(n) + q\phi(n)} = (a^{\phi(n)})^q \cdot a^{k \bmod \phi(n)} \equiv a^{k \bmod \phi(n)} \pmod{n}$

Fermat's Little Theorem (Special case of Euler's):

Theorem: If p is prime and $\gcd(a, p) = 1$, then:

$a^{p-1} \equiv 1 \pmod{p}$

Equivalently (for any a): $a^p \equiv a \pmod{p}$

Proof 1 (Euler's theorem): $\phi(p) = p - 1$ for prime p

Proof 2 (Combinatorial):

- Consider necklaces with p beads, each colored one of a colors
- Total: a^p necklaces
- Fixed under rotation: Only monochrome necklaces (a of them)
- Necklaces not fixed form orbits of size p under rotation
- So $a^p \equiv a \pmod{p}$

Proof 3 (Elementary):

- Consider $a, 2a, 3a, \ldots, (p-1)a$ modulo p
- All distinct mod p (since $\gcd(a, p) = 1$)

- So they're a permutation of $1, 2, 3, \ldots, p-1$
- Product: $a \cdot 2a \cdot 3a \cdots (p-1)a \equiv 1 \cdot 2 \cdot 3 \cdots (p-1) \pmod{p}$
- $a^{p-1}(p-1)! \equiv (p-1)! \pmod{p}$
- Cancel $(p-1)!$: $a^{p-1} \equiv 1 \pmod{p}$

Applications:

1. Primality testing (Fermat test):

- If $a^{n-1} \not\equiv 1 \pmod{n}$ for some a, then n is composite
- Converse not always true (Carmichael numbers!)

2. Computing inverses mod prime:

- $a^{-1} \equiv a^{p-2} \pmod{p}$ (from $a \cdot a^{p-2} = a^{p-1} \equiv 1$)

3. Computing large powers:

- $2^{100} \bmod 7$: Since $\phi(7) = 6$, $2^{100} = 2^{16 \cdot 6 + 4} \equiv 2^4 = 16 \equiv 2 \pmod{7}$

4. RSA Cryptography:

- Choose primes p, q; let $n = pq$, $\phi(n) = (p-1)(q-1)$
- Public key: (n, e) where $\gcd(e, \phi(n)) = 1$
- Private key: d where $ed \equiv 1 \pmod{\phi(n)}$
- Encrypt: $c = m^e \bmod n$
- Decrypt: $m = c^d \bmod n$
- Correctness: $c^d = (m^e)^d = m^{ed} = m^{1 + k\phi(n)} = m \cdot (m^{\phi(n)})^k \equiv m \pmod{n}$

Problem-Solving Tips:

- Fast exponentiation: Use repeated squaring $O(\log k)$ multiplications
- For modular inverse: Extended Euclid or (for prime p) use $a^{-1} \equiv a^{p-2}$
- Reduce exponents using $\phi(n)$
- CRT for systems with coprime moduli
- GATE: Often test Fermat, totient computation, inverse finding

Examples:

1. **Compute** $7^{222} \bmod 10$:
   - $\phi(10) = \phi(2)\phi(5) = 1 \cdot 4 = 4$
   - $\gcd(7, 10) = 1$, so $7^4 \equiv 1 \pmod{10}$
   - $222 = 55 \cdot 4 + 2$
   - $7^{222} = (7^4)^{55} \cdot 7^2 \equiv 1^{55} \cdot 49 \equiv 9 \pmod{10}$
2. **Solve** $3x \equiv 7 \pmod{11}$:
   - Find $3^{-1} \bmod 11$

- Method 1: $3^{-1} \equiv 3^{11-2} = 3^9 \bmod 11$
  - $3^2 = 9, 3^4 = 81 \equiv 4, 3^8 \equiv 16 \equiv 5, 3^9 \equiv 15 \equiv 4 \pmod{11}$
- Method 2: Extended Euclid gives $4 \cdot 3 - 1 \cdot 11 = 1$, so $3^{-1} \equiv 4$
- Solution: $x \equiv 7 \cdot 4 = 28 \equiv 6 \pmod{11}$

3. **CRT example**: Solve $x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$
   - $M = 3 \cdot 5 \cdot 7 = 105$
   - $M_1 = 35, M_2 = 21, M_3 = 15$
   - $35 \equiv 2 \pmod{3}$, so $y_1 = 2^{-1} \equiv 2 \pmod{3}$
   - $21 \equiv 1 \pmod{5}$, so $y_2 = 1$
   - $15 \equiv 1 \pmod{7}$, so $y_3 = 1$
   - $x \equiv 2(35)(2) + 3(21)(1) + 2(15)(1) = 140 + 63 + 30 = 233 \equiv 23 \pmod{105}$

1.6 Pigeonhole Principle (2) {#16-pigeonhole-principle-2}

Key Concepts: One of the most fundamental principles in combinatorics. Despite its simplicity, it yields powerful existence proofs. The principle guarantees existence without construction.

Basic Pigeonhole Principle (Dirichlet's Box Principle):

Statement: If n objects are placed into m containers (pigeonholes) and $n > m$, then at least one container must contain more than one object.

Formal: If $n > m$, then there exists a pigeonhole with at least $\lceil n/m \rceil$ pigeons.

Proof (by contradiction):

- Assume each pigeonhole contains at most 1 pigeon
- Then total pigeons ≤ m × 1 = m
- But we have n > m pigeons - contradiction!
- Therefore, at least one pigeonhole must contain ≥ 2 pigeons

Generalized Pigeonhole Principle:

Statement: If n objects are placed into m pigeonholes, then:

- At least one pigeonhole contains at least $\lceil n/m \rceil$ objects
- At least one pigeonhole contains at most $\lfloor n/m \rfloor$ objects

Proof:

- Suppose all pigeonholes contain < $\lceil n/m \rceil$ objects
- Then each contains ≤ $\lceil n/m \rceil - 1 = \lfloor n/m \rfloor$ objects
- Total: ≤ $m \cdot \lfloor n/m \rfloor < m \cdot (n/m) = n$ - contradiction!

Strong Form: If $n > km$, then at least one pigeonhole contains at least $k + 1$ objects.

Proof: If all pigeonholes had ≤ k objects, total would be ≤ km < n - contradiction.

Applications and Classic Problems:

1. Sock Drawer Problem:

- 10 pairs of socks (20 socks total) in a dark drawer
- How many socks must you pull out to guarantee a matching pair?
- Answer: 11 (10 colors + 1)
- Pigeons: socks pulled out
- Pigeonholes: colors (10)
- By PHP: If pulling 11 socks, at least one color appears twice

2. Birthday Problem (Existence version):

- In any group of 367 people, at least two share a birthday
- Pigeons: 367 people
- Pigeonholes: 366 possible birthdays (including Feb 29)
- 367 > 366 → PHP applies

3. Subset Sum Problem:

- Given any n+1 integers from $\{1, 2, \ldots, 2n\}$, there exist two whose sum is $2n + 1$
- Proof: Pair integers: $(1, 2n), (2, 2n - 1), \ldots, (n, n + 1)$
- n pairs (pigeonholes), n+1 integers (pigeons)
- Two integers from same pair sum to $2n + 1$

4. Ramsey Theory (R(3,3) = 6):

- In any group of 6 people, there exist 3 mutual friends OR 3 mutual strangers
- Model as graph coloring problem
- Consider person A: They know or don't know each of 5 others
- By PHP: At least 3 relationships of same type (say, knows B, C, D)
- If any of B,C,D know each other → 3 mutual friends
- If none of B,C,D know each other → 3 mutual strangers

5. Decimal Expansion:

- Every rational number has eventually repeating decimal expansion
- When dividing, remainders are from $\{0, 1, \ldots, d - 1\}$ (d pigeonholes)
- After d+1 divisions (pigeons), by PHP, a remainder repeats
- This forces cycle in decimal expansion

6. Sequence Problem:

- Among any n+1 positive integers ≤ 2n, there exist two where one divides the other

- Proof: Write each as $2^k \cdot m$ where m is odd
- Only n odd numbers ≤ 2n
- By PHP on n+1 integers: Two have same odd part m
- Say $2^{k_1}m$ and $2^{k_2}m$ - one divides the other

Average Argument (Probabilistic PHP):

Principle: If the average value is x, then:

- At least one value is ≥ x
- At least one value is ≤ x

Example: In a graph with n vertices and e edges, there exists:

- A vertex with degree ≥ $2e/n$ (average degree)
- A vertex with degree ≤ $2e/n$

Infinite Pigeonhole Principle:

Statement: If infinitely many objects are placed in finitely many pigeonholes, at least one pigeonhole contains infinitely many objects.

Application - Bolzano-Weierstrass: Every bounded infinite sequence has a convergent subsequence.

Erdős-Szekeres Theorem (Monotonic Subsequences):

Theorem: Any sequence of $n^2 + 1$ distinct real numbers contains either:

- An increasing subsequence of length n+1, OR
- A decreasing subsequence of length n+1

Proof (via PHP):

- Assign each element a_i a pair (L_i, D_i):
  - L_i = length of longest increasing subsequence ending at a_i
  - D_i = length of longest decreasing subsequence ending at a_i
- If all pairs have $L_i, D_i \leq n$, there are at most n^2 possible pairs
- With $n^2 + 1$ elements (pigeons) and n^2 pairs (pigeonholes)
- By PHP: Two elements a_i, a_j have same pair
- If $i < j$ and $a_i < a_j$: Can extend increasing subseq - contradiction
- If $i < j$ and $a_i > a_j$: Can extend decreasing subseq - contradiction
- Therefore, assumption fails: some L_i or D_i exceeds n

Problem-Solving Strategy:

1. **Identify pigeons**: Objects to be distributed

2. **Identify pigeonholes**: Categories/containers
3. **Count carefully**: Ensure n > m (or n > km for strong form)
4. **Extract conclusion**: "At least one pigeonhole contains..."
5. **For impossibility**: Show any distribution violates PHP

GATE Tips:

- PHP proves existence, not construction
- Often combined with other principles (extremal principle, induction)
- Watch for "at least" vs "at most" conclusions
- Average argument useful for graph theory
- Strong form: To guarantee k+1 items in one hole, need km+1 total items

Examples:

1. **13 pigeons, 12 holes**: At least one hole has ≥ $\lceil 13/12 \rceil = 2$ pigeons
2. **Prove**: Among any 5 integers, two differ by a multiple of 4
   - Remainders mod 4: {0,1,2,3} (4 pigeonholes)
   - 5 integers (pigeons)
   - By PHP: Two have same remainder mod 4
   - Their difference is divisible by 4
3. **Social network**: In any group of n people, at least two have the same number of friends within the group
   - Possible friend counts: 0, 1, 2, ..., n-1
   - But 0 and n-1 can't coexist (if someone has 0 friends, no one has n-1)
   - So at most n-1 possible values (pigeonholes)
   - n people (pigeons)
   - By PHP: Two people have same friend count
4. **Geometric**: Place 5 points inside a unit square. Prove two points are within distance $\frac{\sqrt{2}}{2}$
   - Divide square into 4 equal subsquares (side = 1/2)
   - 5 points (pigeons), 4 subsquares (pigeonholes)
   - By PHP: One subsquare contains ≥2 points
   - Max distance in subsquare: diagonal = $\frac{\sqrt{2}}{2}$

1.7 Recurrence Relation (7) {#17-recurrence-relation-7}

Key Concepts: A recurrence relation defines a sequence recursively - each term as a function of previous terms. Essential for analyzing algorithms and counting problems.

Definition: A recurrence relation for sequence $\{a_n\}$ expresses a_n in terms of previous terms $a_0, a_1, \ldots, a_{n-1}$, along with initial conditions.

Types of Recurrence Relations:

1. Linear vs Nonlinear:

- **Linear**: Each term appears to first power: $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots$
- **Nonlinear**: Products, powers of terms: $a_n = a_{n-1}^2$, $a_n = a_{n-1} \cdot a_{n-2}$

2. Homogeneous vs Non-homogeneous:

- **Homogeneous**: RHS only contains previous terms: $a_n = 2a_{n-1} + 3a_{n-2}$
- **Non-homogeneous**: RHS has additional function: $a_n = 2a_{n-1} + n$

3. Order: Highest index difference

- $a_n = a_{n-1} + a_{n-2}$ is 2nd order
- $a_n = a_{n-1} + a_{n-3}$ is 3rd order

Linear Homogeneous Recurrence Relations (LHRR):

General Form: $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$

Solution Method - Characteristic Equation:

Step 1: Assume solution of form $a_n = r^n$

Step 2: Substitute into recurrence:

$r^n = c_1 r^{n-1} + c_2 r^{n-2} + \cdots + c_k r^{n-k}$

Divide by r^{n-k}:

$r^k = c_1 r^{k-1} + c_2 r^{k-2} + \cdots + c_k$

Step 3: Rearrange to **characteristic equation**:

$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_k = 0$

Step 4: Find roots $r_1, r_2, \ldots, r_k$

Step 5: Construct general solution based on root types:

Case 1: Distinct Real Roots $r_1, r_2, \ldots, r_k$ all different:

$a_n = A_1 r_1^n + A_2 r_2^n + \cdots + A_k r_k^n$

Case 2: Repeated Roots - Root r with multiplicity m:

$a_n = (A_1 + A_2 n + A_3 n^2 + \cdots + A_m n^{m-1}) r^n$

Case 3: Complex Roots $r = \rho e^{i\theta} = \rho(\cos\theta + i\sin\theta)$:

$a_n = \rho^n (A\cos(n\theta) + B\sin(n\theta))$

Step 6: Use initial conditions to find constants $A_1, A_2, \ldots$

Classic Examples:

Fibonacci Sequence: $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0, F_1 = 1$

Solution:

1. Characteristic equation: $r^2 = r + 1 \rightarrow r^2 - r - 1 = 0$
2. Roots (quadratic formula): $r = \frac{1 \pm \sqrt{5}}{2}$
   - $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ (golden ratio)
   - $\psi = \frac{1-\sqrt{5}}{2} \approx -0.618$
3. General solution: $F_n = A\phi^n + B\psi^n$
4. Initial conditions:
   - $F_0 = 0 : A + B = 0 \rightarrow B = -A$
   - $F_1 = 1 : A\phi + B\psi = 1$
   - Solving: $A = \frac{1}{\sqrt{5}}, B = -\frac{1}{\sqrt{5}}$
5. **Binet's Formula**: $F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}$

Properties:

- Since $|\psi| < 1$, $\psi^n \rightarrow 0$
- $F_n \approx \frac{\phi^n}{\sqrt{5}}$ (nearest integer)
- $\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \phi$

Tower of Hanoi: $T_n = 2T_{n-1} + 1$, $T_1 = 1$

Homogeneous part: $T_n^{(h)} = 2T_{n-1} \rightarrow r = 2 \rightarrow T_n^{(h)} = A \cdot 2^n$

Particular solution: Try constant $T_n^{(p)} = c$

- $c = 2c + 1 \rightarrow c = -1$

General: $T_n = A \cdot 2^n - 1$

Initial condition: $T_1 = 1 \rightarrow 2A - 1 = 1 \rightarrow A = 1$

Solution: $T_n = 2^n - 1$

Non-homogeneous Linear Recurrences:

Form: $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k} + f(n)$

Solution: $a_n = a_n^{(h)} + a_n^{(p)}$

- $a_n^{(h)}$: Homogeneous solution (solve characteristic equation)
- $a_n^{(p)}$: Particular solution (guess based on $f(n)$)

Particular Solution Guesses:

- $f(n) = \text{polynomial of degree d}$ → Try polynomial of degree d
- $f(n) = \alpha^n$ → Try $c \cdot \alpha^n$ (unless α is root of char. eq.)
- $f(n) = \alpha^n \cdot p(n)$ → Try $\alpha^n \cdot q(n)$ where deg(q) = deg(p)

If guess form matches homogeneous solution: Multiply by n^m where m is multiplicity

Divide and Conquer Recurrences:

Form: $T(n) = aT(n/b) + f(n)$

- a: number of subproblems
- n/b: size of each subproblem
- f(n): work outside recursive calls

Master Theorem: For $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$:

Let $c_{crit} = \log_b a$

Case 1: If $f(n) = O(n^c)$ for $c < c_{crit}$:

- $T(n) = \Theta(n^{c_{crit}})$
- Tree-depth dominates

Case 2: If $f(n) = \Theta(n^{c_{crit}} \log^k n)$:

- $T(n) = \Theta(n^{c_{crit}} \log^{k+1} n)$
- All levels contribute equally

Case 3: If $f(n) = \Omega(n^c)$ for $c > c_{crit}$ AND regularity condition:

- $T(n) = \Theta(f(n))$
- Root dominates

Examples:

1. **Merge Sort**: $T(n) = 2T(n/2) + n$
   - $a = 2, b = 2, f(n) = n$
   - $c_{crit} = \log_2 2 = 1$
   - $f(n) = \Theta(n^1)$ → Case 2 with k=0
   - $T(n) = \Theta(n \log n)$
2. **Binary Search**: $T(n) = T(n/2) + O(1)$
   - $a = 1, b = 2, f(n) = 1$
   - $c_{crit} = 0$
   - Case 2: $T(n) = \Theta(\log n)$
3. **Karatsuba Multiplication**: $T(n) = 3T(n/2) + O(n)$
   - $c_{crit} = \log_2 3 \approx 1.585$

- $f(n) = O(n^1)$, $1 < 1.585$
- Case 1: $T(n) = \Theta(n^{1.585})$

Generating Function Method (see section 1.4):

Define $G(x) = \sum_{n=0}^{\infty} a_n x^n$, manipulate recurrence algebraically

Substitution/Iteration Method:

Expand recurrence repeatedly:

- $a_n = f(a_{n-1}) = f(f(a_{n-2})) = \cdots$
- Look for pattern
- Prove by induction

GATE Tips:

- Identify recurrence type (linear homogeneous, etc.)
- For LHRR: Characteristic equation is key
- Master theorem: Compare $f(n)$ with $n^{\log_b a}$
- Common mistake: Forgetting initial conditions for constants
- Fibonacci appears frequently - memorize Binet's formula

Examples:

1. Solve $a_n = 2a_{n-1}, a_0 = 1$:
   - Char. eq: $r = 2$
   - Solution: $a_n = A \cdot 2^n$
   - $a_0 = 1 : A = 1$
   - $a_n = 2^n$
2. Solve $a_n = 6a_{n-1} - 9a_{n-2}, a_0 = 1, a_1 = 3$:
   - Char. eq: $r^2 - 6r + 9 = 0 \rightarrow (r-3)^2 = 0$
   - Repeated root $r = 3$ (multiplicity 2)
   - $a_n = (A + Bn) \cdot 3^n$
   - $a_0 = 1 : A = 1$
   - $a_1 = 3 : 3(1 + B) = 3 \rightarrow B = 0$
   - $a_n = 3^n$
3. Catalan numbers: $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}, C_0 = 1$:
   - Generating function technique gives: $C_n = \frac{1}{n+1}\binom{2n}{n}$

1.8 Summation (3) {#18-summation-3}

Key Concepts: Finding closed-form expressions for sums. Essential for algorithm analysis, probability calculations, and mathematical proofs.

Fundamental Summation Formulas:

1. Sum of First n Natural Numbers:

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

Derivation (Gauss method):

- Write sum forward: $S = 1 + 2 + 3 + \cdots + n$
- Write sum backward: $S = n + (n-1) + (n-2) + \cdots + 1$
- Add vertically: $2S = (n+1) + (n+1) + \cdots + (n+1)$ (n times)
- $2S = n(n+1)$
- $S = \frac{n(n+1)}{2}$

Alternative derivation (Visual/combinatorial):

- Counting pairs: Choose 2 from n+1 elements including 0
- Total arrangements in triangle pattern

2. Sum of First n Squares:

$\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$

Derivation (Using telescoping):

- Consider $(k+1)^3 - k^3 = 3k^2 + 3k + 1$
- Sum from k=1 to n: $\sum_{k=1}^{n}[(k+1)^3 - k^3] = \sum_{k=1}^{n}(3k^2 + 3k + 1)$
- LHS telescopes: $(n+1)^3 - 1^3 = n^3 + 3n^2 + 3n$
- RHS: $3\sum k^2 + 3\sum k + n$
- $n^3 + 3n^2 + 3n = 3\sum k^2 + 3 \cdot \frac{n(n+1)}{2} + n$
- Solve for $\sum k^2$: $\sum k^2 = \frac{n^3+3n^2+3n-\frac{3n(n+1)}{2}-n}{3} = \frac{n(n+1)(2n+1)}{6}$

3. Sum of First n Cubes:

$\sum_{k=1}^{n} k^3 = 1^3 + 2^3 + 3^3 + \cdots + n^3 = \left[\frac{n(n+1)}{2}\right]^2$

Remarkable property: Sum of cubes = Square of sum!

$\sum_{k=1}^{n} k^3 = \left(\sum_{k=1}^{n} k\right)^2$

Proof (Induction):

- Base: n=1: $1^3 = 1^2$ ✓
- Assume true for n: $\sum_{k=1}^{n} k^3 = \left[\frac{n(n+1)}{2}\right]^2$
- Prove for n+1: $\sum_{k=1}^{n+1} k^3 = \left[\frac{n(n+1)}{2}\right]^2 + (n+1)^3 = \frac{n^2(n+1)^2+4(n+1)^3}{4} = \frac{(n+1)^2[n^2+4(n+1)]}{4}$
  $= \frac{(n+1)^2(n+2)^2}{4} = \left[\frac{(n+1)(n+2)}{2}\right]^2$ ✓

4. Geometric Series:

$\sum_{k=0}^{n} r^k = 1 + r + r^2 + \cdots + r^n = \begin{cases} \frac{r^{n+1}-1}{r-1} & \text{if } r \neq 1 \\ n+1 & \text{if } r = 1 \end{cases}$

Derivation:

- Let $S = 1 + r + r^2 + \cdots + r^n$
- Multiply by r: $rS = r + r^2 + r^3 + \cdots + r^{n+1}$
- Subtract: $S - rS = 1 - r^{n+1}$
- $S(1-r) = 1 - r^{n+1}$
- $S = \frac{1-r^{n+1}}{1-r} = \frac{r^{n+1}-1}{r-1}$ (multiply by -1/-1)

Infinite geometric series ($|r| < 1$):

$\sum_{k=0}^{\infty} r^k = \frac{1}{1-r}$ (as $r^{n+1} \to 0$)

5. Arithmetic-Geometric Series:

$\sum_{k=0}^{n} k \cdot r^k = 0 + r + 2r^2 + 3r^3 + \cdots + nr^n = \frac{r(1-(n+1)r^n+nr^{n+1})}{(1-r)^2}$

Derivation (Differentiation trick):

- Start with $\sum_{k=0}^{n} r^k = \frac{1-r^{n+1}}{1-r}$
- Differentiate both sides with respect to r
- Apply product and chain rules

6. Harmonic Series:

$H_n = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

No closed form, but approximation:

$H_n \approx \ln(n) + \gamma$ where $\gamma \approx 0.5772$ (Euler-Mascheroni constant)

Growth: $H_n = \Theta(\log n)$

7. Power Sums (General):

$\sum_{k=1}^{n} k^p$ can be expressed as polynomial of degree p+1 in n

Faulhaber's Formulas: Use Bernoulli numbers

- For p=0: $\sum 1 = n$
- For p=1: $\sum k = \frac{n^2}{2} + \frac{n}{2}$
- For p=2: $\sum k^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$
- For p=3: $\sum k^3 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$

Summation Techniques:

1. Telescoping Series:

- Series where consecutive terms cancel
- $\sum_{k=1}^{n}[f(k+1)-f(k)]=f(n+1)-f(1)$

Example: $\sum_{k=1}^{n}\frac{1}{k(k+1)}$

- Partial fractions: $\frac{1}{k(k+1)}=\frac{1}{k}-\frac{1}{k+1}$
- Sum: $\left(\frac{1}{1}-\frac{1}{2}\right)+\left(\frac{1}{2}-\frac{1}{3}\right)+\cdots+\left(\frac{1}{n}-\frac{1}{n+1}\right)$
- Most terms cancel: $1-\frac{1}{n+1}=\frac{n}{n+1}$

2. Perturbation Method:

- Compare S with rS (for geometric-like series)
- Used in geometric series derivation

3. Differentiation/Integration:

- For $\sum k\cdot r^k$, differentiate $\sum r^k$
- For $\sum\frac{r^k}{k}$, integrate $\sum r^{k-1}$

4. Generating Functions (see section 1.4):

- Encode sum as power series
- Manipulate algebraically
- Extract coefficient

5. Mathematical Induction:

- Base case: Verify for n=1
- Inductive step: Assume for n, prove for n+1
- Essential for proving formulas

Double Summations:

Order exchange (Fubini's theorem for finite sums):

$\sum_{i=1}^{m}\sum_{j=1}^{n}a_{ij}=\sum_{j=1}^{n}\sum_{i=1}^{m}a_{ij}$

Triangular summation:

$\sum_{1\le i<j\le n}a_{ij}$ (sum over pairs)

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

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

Binomial Sum Identities:

1. $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ (sum of binomial coefficients)
   - Proof: Expand $(1+1)^n$
2. $\sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0$ (alternating sum)
   - Proof: Expand $(1-1)^n = 0$ for n≥1
3. $\sum_{k=0}^{n} k\binom{n}{k} = n \cdot 2^{n-1}$
   - Proof: Differentiate $(1+x)^n$, set x=1
4. $\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}$ (Vandermonde's identity)

Problem-Solving Strategy:

1. Check if it's a standard formula (arithmetic, geometric, powers)
2. Try telescoping (partial fractions for rational terms)
3. For products with powers, try perturbation method
4. For complex sums, try generating functions
5. Verify with induction after finding pattern

GATE Tips:

- Memorize formulas for $\sum k, \sum k^2, \sum k^3$, geometric series
- Sum of first n odd numbers: $1 + 3 + 5 + \cdots + (2n-1) = n^2$
- Sum of first n even numbers: $2 + 4 + 6 + \cdots + 2n = n(n+1)$
- Telescoping: Look for differences
- Geometric series: Check common ratio
- Double sums: Try changing order or substitution
- For algorithm analysis: Often need $\sum k$ or geometric sums

Common Patterns:

- $\sum_{k=1}^{n} (2k-1) = n^2$ (odd numbers)
- $\sum_{k=1}^{n} k^2 - (k-1)^2 = n^2$ (telescoping)
- $\sum_{k=0}^{n} k \cdot 2^k = (n-1)2^{n+1} + 2$ (arithmetic-geometric)

Examples:

1. **Find** $\sum_{k=1}^{100} (3k-1)$:
   - $3\sum k - \sum 1 = 3 \cdot \frac{100 \cdot 101}{2} - 100 = 15150 - 100 = 15050$
2. **Compute** $\sum_{k=1}^{n} k(k+1)$:
   - $\sum (k^2 + k) = \sum k^2 + \sum k = \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}$
   - $= \frac{n(n+1)}{6}[2n+1+3] = \frac{n(n+1)(2n+4)}{6} = \frac{n(n+1)(n+2)}{3}$
3. **Evaluate** $\sum_{k=1}^{\infty} \frac{1}{2^k}$:
   - Geometric with r=1/2: $\sum_{k=1}^{\infty} \left(\frac{1}{2}\right)^k = \frac{1/2}{1-1/2} = 1$
4. **Prove**: $1^3 + 2^3 + \cdots + n^3 = (1 + 2 + \cdots + n)^2$:
   - LHS: $\sum k^3 = \left[\frac{n(n+1)}{2}\right]^2$

- RHS: $\left(\sum k\right)^2 = \left[\frac{n(n+1)}{2}\right]^2$
- Equal! ✓

2. Discrete Mathematics: Graph Theory (83)

2.1 Counting (3) {#21-counting-3}

Key Concepts: Enumeration problems in graph theory - counting specific graph structures and properties.

Trees:

- **Definition**: Connected acyclic graph
- **Properties**:
  - n vertices → exactly n-1 edges
  - Any two vertices connected by unique path
  - Adding any edge creates exactly one cycle
  - Removing any edge disconnects the graph
  - Minimum edges for connectivity

Cayley's Formula (Labeled Trees):

Number of labeled trees on n vertices: n^{n-2}

Proof Sketch (Prüfer Sequence):

- Each labeled tree on n vertices corresponds to unique sequence of length n-2
- Each element in sequence is from {1,2,...,n}
- Total sequences: n^{n-2}
- Bijection establishes formula

Examples:

- n=2: $2^0 = 1$ tree
- n=3: $3^1 = 3$ trees
- n=4: $4^2 = 16$ trees

Spanning Trees (Kirchhoff's Matrix-Tree Theorem):

For connected graph G:

- Construct Laplacian matrix L = D - A
  - D = degree matrix (diagonal)
  - A = adjacency matrix
- Number of spanning trees = any cofactor of L

- Equivalently: Determinant of any (n-1)×(n-1) submatrix obtained by deleting one row and corresponding column

Example - K_3 (Complete graph on 3 vertices):

- Laplacian: $L = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$
- Delete row 1, column 1: $\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$
- Determinant: $2(2) - (-1)(-1) = 4 - 1 = 3$
- Answer: 3 spanning trees

Complete Graph K_n:

Number of spanning trees in K_n: n^{n-2} (Cayley's formula)

Complete Bipartite Graph $K_{m,n}$:

Number of spanning trees: $m^{n-1} \cdot n^{m-1}$

Eulerian Paths and Circuits:

Eulerian Circuit: Path that visits every edge exactly once and returns to start

- **Exists iff**: Graph is connected AND all vertices have even degree
- **Count**: No simple closed formula (computationally hard)

Eulerian Path: Path that visits every edge exactly once

- **Exists iff**: Graph is connected AND exactly 0 or 2 vertices have odd degree
- If 2 odd-degree vertices: Path starts at one, ends at other

Hamiltonian Paths and Cycles:

Hamiltonian Cycle: Path that visits every vertex exactly once and returns to start

- **No simple criterion** for existence (NP-complete)
- **Sufficient conditions**:
  - Dirac: If deg(v) ≥ n/2 for all v, then Hamiltonian
  - Ore: If deg(u) + deg(v) ≥ n for all non-adjacent u,v, then Hamiltonian

Hamiltonian Path: Path that visits every vertex exactly once

- Also NP-complete to determine existence

Counting in Specific Graphs:

1. Perfect Matchings:

- In K_{2n}: $(2n-1)!! = (2n-1)(2n-3)\cdots(3)(1)$
- Double factorial notation

2. Chromatic Polynomial $P(G,k)$:

- Number of proper k-colorings of G
- For tree on n vertices: $P(T,k) = k(k-1)^{n-1}$
- For K_n: $P(K_n,k) = k(k-1)(k-2)\cdots(k-n+1)$
- For cycle C_n: $P(C_n,k) = (k-1)^n + (-1)^n(k-1)$

3. Number of Walks:

- Number of walks of length k from vertex i to vertex j: $(A^k)_{ij}$
- A = adjacency matrix
- Total walks of length k: $\text{trace}(A^k)$

Problem-Solving Strategy:

1. For spanning trees: Use Kirchhoff's theorem (small graphs) or formulas (special graphs)
2. For Eulerian: Check degree parity
3. For Hamiltonian: Use sufficient conditions or exhaustive search
4. For labeled counting: Consider Cayley-type formulas

GATE Tips:

- Tree with n vertices has n-1 edges (memorize!)
- K_n has n^{n-2} spanning trees
- Eulerian circuit: All even degrees
- Hamiltonian: No simple test (but know Dirac/Ore conditions)
- Matrix-tree theorem useful for small graphs

Examples:

1. **How many spanning trees in K_4?**
   - By Cayley: $4^{4-2} = 4^2 = 16$
2. **Does C_5 have Eulerian circuit?**
   - All vertices have degree 2 (even) ✓
   - Connected ✓
   - Yes, Eulerian circuit exists
3. **Number of labeled trees on 5 vertices:**
   - $5^{5-2} = 5^3 = 125$
4. **Spanning trees in cycle C_n:**
   - Remove any one edge from cycle → spanning tree
   - Answer: n spanning trees

2.2 Degree of Graph (12) {#22-degree-of-graph-12}

Key Concepts: The degree of a vertex is the number of edges incident to it. Fundamental for characterizing graph properties and proving theorems.

Degree Definitions:

For Undirected Graphs:

- **Degree** deg(v): Number of edges incident to vertex v
- **Loop**: An edge from v to itself contributes 2 to deg(v)
- **Isolated vertex**: deg(v) = 0
- **Pendant/Leaf vertex**: deg(v) = 1

For Directed Graphs:

- **In-degree** $\deg^-(v)$: Number of edges entering v
- **Out-degree** $\deg^+(v)$: Number of edges leaving v
- **Total degree**: $\deg(v) = \deg^-(v) + \deg^+(v)$

Handshaking Lemma (Fundamental Theorem):

Statement: In any undirected graph G = (V, E):

$\sum_{v \in V} \deg(v) = 2|E|$

The sum of all vertex degrees equals twice the number of edges.

Proof:

- Each edge e = {u,v} contributes exactly 1 to deg(u) and 1 to deg(v)
- Therefore, each edge contributes exactly 2 to the total sum
- With |E| edges, total contribution = 2|E|
- QED

Alternative proof (Counting argument):

- Count pairs (v, e) where vertex v is incident to edge e
- From vertex perspective: Each vertex v contributes deg(v) such pairs
- Total: $\sum_v \deg(v)$
- From edge perspective: Each edge contributes 2 such pairs (its two endpoints)
- Total: $2|E|$
- These count the same set, so they're equal

Corollaries:

1. Number of Odd-Degree Vertices is Even:

- Let O = vertices with odd degree, E = vertices with even degree
- $\sum_{v \in O} \deg(v) + \sum_{v \in E} \deg(v) = 2|E|$
- RHS is even
- Second sum is even (sum of even numbers)
- Therefore, first sum must be even
- Sum of |O| odd numbers is even ⟺ |O| is even
- **Conclusion**: In any graph, the number of odd-degree vertices is even

2. Average Degree:

$\bar{d} = \frac{1}{|V|} \sum_v \deg(v) = \frac{2|E|}{|V|}$

For Directed Graphs:

$\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = |E|$

Each edge contributes 1 to in-degree sum and 1 to out-degree sum.

Degree Sequences:

Definition: List of vertex degrees in non-increasing order

- Example: Graph with degrees 3,3,2,2,2,0 has degree sequence (3,3,2,2,2,0)

Graphic Sequence: A sequence that can be realized as the degree sequence of some simple graph

Necessary Conditions (for sequence to be graphic):

1. Sum must be even (by handshaking lemma)
2. Each element ≤ n-1 (max degree in simple graph on n vertices)
3. If sorted as $d_1 \geq d_2 \geq ... \geq d_n$, then $d_1 \leq n-1$

Erdős-Gallai Theorem: Sequence $(d_1, d_2, ..., d_n)$ with $d_1 \geq d_2 \geq ... \geq d_n$ is graphic iff:

1. $\sum_{i=1}^{n} d_i$ is even
2. For each k ∈ {1,...,n}:

   $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)$

Havel-Hakimi Algorithm (Constructive test):

Given sequence S = $(d_1, d_2, ..., d_n)$ sorted in non-increasing order:

1. Remove d_1 from sequence
2. Subtract 1 from the next d_1 largest elements
3. Reorder resulting sequence
4. Repeat until:

- All zeros (graphic!) OR
- Negative number or impossible subtraction (not graphic)

Example: Is (4,3,3,2,2) graphic?

- Remove 4, subtract from next 4: (3,3,2,2) → (2,2,1,1)
- Remove 2, subtract from next 2: (2,1,1) → (0,0)
- All zeros ✓ Graphic!

Regular Graphs:

Definition: A graph is **k-regular** if every vertex has degree k

- All vertices have the same degree

Properties:

1. For k-regular graph on n vertices:
   - By handshaking: $nk = 2|E|$
   - Therefore: $|E| = \frac{nk}{2}$
   - **Consequence**: For k-regular graph to exist, nk must be even
2. **0-regular**: No edges (empty graph)
3. **1-regular**: Perfect matching (n must be even)
4. **2-regular**: Disjoint union of cycles
5. **(n-1)-regular**: Complete graph K_n

Examples of Regular Graphs:

- **Cycle** C_n: 2-regular
- **Complete graph** K_n: (n-1)-regular
- **Complete bipartite** $K_{n,n}$: n-regular
- **Hypercube** Q_n: n-regular (n-dimensional)
- **Petersen graph**: 3-regular (10 vertices)

Degree Bounds and Inequalities:

1. Maximum Degree Δ(G):

- Δ(G) = max{deg(v) : v ∈ V}
- In simple graph: Δ(G) ≤ n-1

2. Minimum Degree δ(G):

- δ(G) = min{deg(v) : v ∈ V}

3. Relationship to Edges:

- $\delta(G) \leq \frac{2|E|}{|V|} \leq \Delta(G)$
- Follows from handshaking lemma

4. For Connected Graphs:

- δ(G) ≥ 1 (no isolated vertices)

Degree and Graph Properties:

1. Eulerian Graphs:

- Has Eulerian circuit ⟺ Connected AND all vertices have even degree
- Has Eulerian path ⟺ Connected AND exactly 0 or 2 vertices have odd degree

2. Hamiltonian Graphs (Sufficient conditions):

- **Dirac's Theorem**: If δ(G) ≥ n/2 and n ≥ 3, then G is Hamiltonian
- **Ore's Theorem**: If deg(u) + deg(v) ≥ n for all non-adjacent u,v, then G is Hamiltonian

3. Connectivity:

- If δ(G) ≥ k, then G has a path of length at least k
- If δ(G) ≥ 2, then G contains a cycle

4. Graph Coloring:

- **Brooks' Theorem**: If G is connected, not complete, and not an odd cycle, then χ(G) ≤ Δ(G)
- Greedy coloring uses at most Δ(G) + 1 colors

5. Trees:

- Connected acyclic graph on n vertices has exactly n-1 edges
- By handshaking: $\sum \deg(v) = 2(n-1)$
- Therefore: Average degree = $\frac{2(n-1)}{n} < 2$
- **Conclusion**: Every tree with n ≥ 2 has at least 2 leaves (deg 1)

Degree Sum Formula for Directed Graphs:

For directed graph:

- $\sum_v \deg^+(v) = \sum_v \deg^-(v) = |E|$
- Each directed edge contributes 1 to exactly one out-degree and one in-degree

Tournament: Complete directed graph (one direction for each pair)

- For tournament on n vertices: $\sum \deg^+(v) = \binom{n}{2}$

Degree Centrality (Network Analysis):

Normalized degree centrality:

$C_D(v) = \frac{\deg(v)}{n-1}$

Measures importance of vertex in network (0 to 1 scale)

Problem-Solving Strategy:

1. Use handshaking lemma to find |E| from degree sum
2. Check degree sequence realizability with Havel-Hakimi
3. Count odd-degree vertices (must be even!)
4. For regular graphs: Check if nk is even
5. Apply degree-based theorems (Dirac, Ore, Brooks)

GATE Tips:

- Handshaking: $\sum \deg = 2|E|$ (most frequently used formula)
- Number of odd-degree vertices is always even
- Tree on n vertices: sum of degrees = 2(n-1)
- k-regular on n vertices: nk must be even for graph to exist
- Complete graph K_n: Each vertex has degree n-1
- For Eulerian: Check all degrees even (circuit) or exactly 2 odd (path)
- Degree sequence: Sum must be even, each element ≤ n-1

Common Mistakes:

- Forgetting loops count twice toward degree
- Confusing maximum degree with number of vertices
- Not checking if nk is even for regular graphs
- Assuming any even-sum sequence is graphic

Examples:

1. **Graph with 10 edges, 5 vertices. If 4 vertices have degree 3, what is the degree of the 5th vertex?**
   - By handshaking: $\sum \deg = 2 \times 10 = 20$
   - First 4 vertices contribute: $4 \times 3 = 12$
   - Fifth vertex: $20 - 12 = 8$
   - But wait! Maximum degree in simple graph with 5 vertices is 4
   - Answer: Not possible as simple graph (or has multiple edges)
2. **Is (5,4,4,3,2,2) a graphic sequence?**
   - Sum = 20 (even) ✓
   - Havel-Hakimi:

        - Remove 5, subtract: (4,4,3,2,2) → (3,3,2,1,1)
        - Remove 3, subtract: (3,2,1,1) → (1,0,0)
        - Remove 1, subtract: (0,0) → (-1,0)
    - Negative! Not graphic ✗
3. **How many edges in a 4-regular graph on 7 vertices?**
    - $|E| = \frac{nk}{2} = \frac{7 \times 4}{2} = 14$
4. **Prove: Every tree with n ≥ 2 vertices has at least 2 leaves.**
    - Proof: Tree has n-1 edges
    - By handshaking: $\sum \deg = 2(n-1)$
    - If ≤ 1 leaf, then at most 1 vertex with deg(v) = 1
    - All other n-1 vertices have deg(v) ≥ 2
    - Sum: $\sum \deg \geq 1 + 2(n-1) = 2n - 1$
    - But $\sum \deg = 2n - 2$
    - Contradiction! Must have ≥ 2 leaves ✓

2.3 Graph Coloring (11) {#23-graph-coloring-11}

Key Concepts: Assign colors to vertices so no adjacent vertices share color. Minimum colors needed = chromatic number χ(G).

Chromatic Number χ(G):

- Minimum number of colors needed for proper vertex coloring
- NP-complete to compute for general graphs
- **Bounds**: $\omega(G) \leq \chi(G) \leq \Delta(G) + 1$
    - ω(G) = clique number (size of largest complete subgraph)
    - Δ(G) = maximum degree

Special Cases:

- **Empty graph**: χ = 1
- **Tree**: χ = 2 (bipartite)
- **Bipartite graph**: χ = 2 (no odd cycles)
- **Complete graph** K_n: χ = n
- **Cycle** C_n:
    - If n even: χ = 2
    - If n odd: χ = 3
- **Wheel** W_n:
    - If n even: χ = 3
    - If n odd: χ = 4
- **Planar graph**: χ ≤ 4 (Four Color Theorem)

Important Theorems:

1. Brooks' Theorem:

- For connected graph G that is neither complete nor odd cycle: $\chi(G) \leq \Delta(G)$
- Exception: Complete graphs and odd cycles need $\Delta(G) + 1$ colors
- Regular graph: If not complete/odd cycle, can color with degree colors

2. Four Color Theorem:

- Every planar graph can be colored with ≤ 4 colors
- Proof: Computer-assisted (controversial initially)
- Practical: Many planar graphs need only 3 colors

3. Five Color Theorem:

- Every planar graph can be colored with ≤ 5 colors
- Easier proof than 4-color
- Proof by induction on vertices

Greedy Coloring Algorithm:

```
Greedy(G):
    Order vertices as v1, v2, ..., vn
    For each vertex vi:
        Color vi with smallest available color
        (not used by any colored neighbor)
```

- **Time**: $O(V + E)$
- **Colors used**: At most $\Delta(G) + 1$
- **Not optimal**: Depends on vertex ordering
- **Welsh-Powell**: Order by degree (decreasing) for better results

k-Colorable Graph:

- Graph that can be properly colored with k colors
- **Decision problem**: "Is χ(G) ≤ k?" is NP-complete for k ≥ 3
- For k = 2: Polynomial time (check bipartiteness via BFS/DFS)

Bipartite Graphs (2-colorable):

- Graph is bipartite ⟺ χ(G) = 2 ⟺ No odd cycles
- **Algorithm**: BFS/DFS with alternating colors
  - If contradiction (neighbor has same color) → Not bipartite
- **Applications**: Matching problems, stable marriage

Edge Coloring:

- Assign colors to edges so adjacent edges have different colors
- **Edge chromatic number χ'(G)**:
  - Minimum colors needed for edge coloring
  - **Vizing's Theorem**: $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$
  - **Class 1**: χ'(G) = Δ(G)
  - **Class 2**: χ'(G) = Δ(G) + 1
- **Bipartite graphs**: Always Class 1 (χ' = Δ)

Graph Coloring Applications:

1. **Register Allocation**: Variables = vertices, conflicts = edges
2. **Scheduling**: Tasks = vertices, conflicts = edges
3. **Frequency Assignment**: Radio stations avoiding interference
4. **Sudoku**: Each cell is vertex, constraints are edges
5. **Map Coloring**: Countries = vertices, borders = edges
6. **Exam Scheduling**: Students = vertices, common courses = edges

Related Problems:

1. Clique:

- Complete subgraph (all pairs connected)
- Clique number ω(G) = size of maximum clique
- ω(G) ≤ χ(G) (need at least ω colors)
- Finding maximum clique is NP-complete

2. Independent Set:

- Set of vertices with no edges between them
- Vertices colored with same color form independent set
- Independence number α(G) = maximum independent set size
- χ(G) ≥ $\frac{|V|}{\alpha(G)}$

3. Vertex Cover:

- Set of vertices covering all edges
- Complement of independent set
- In bipartite graphs: König's theorem

Advanced Concepts:

1. List Coloring:

- Each vertex has list of available colors
- **List chromatic number** $\chi_l(G)$: Minimum k such that any k-lists allow proper coloring

- Always: $\chi_l(G) \geq \chi(G)$

2. Fractional Coloring:

- Generalization allowing fractional colors
- **Fractional chromatic number** $\chi f(G) \leq \chi(G)$

3. Perfect Graphs:

- Graph where $\chi(H) = \omega(H)$ for every induced subgraph H
- **Strong Perfect Graph Theorem**: G is perfect $\iff$ G and complement have no odd cycles ≥ 5
- Includes: Bipartite, chordal, comparability graphs

Problem-Solving Tips:

- Check if bipartite first (BFS with 2 colors)
- For planar: Use ≤ 4 colors
- For trees: Always 2 colors
- Odd cycle: Needs 3 colors minimum
- Complete graph: Needs n colors
- GATE trap: Chromatic number of empty graph is 1, not 0
- Edge coloring: Vizing's theorem gives tight bounds

Verification:

- Check each edge connects different colored vertices
- Count colors used
- Verify optimality (hard in general)

GATE Tips:

- Bipartite $\iff$ 2-colorable $\iff$ No odd cycles
- Planar graphs: $\chi \leq 4$ (4-color theorem)
- Brooks: $\chi \leq \Delta$ (except complete/odd cycle)
- Tree: Always $\chi = 2$
- C_n: $\chi = 2$ if n even, 3 if n odd
- K_n: $\chi = n$
- Greedy uses at most $\Delta+1$ colors
- Clique number $\leq$ Chromatic number $\leq$ Max degree + 1
- Edge chromatic: $\Delta \leq \chi' \leq \Delta+1$ (Vizing)
- NP-complete to decide if $\chi(G) \leq k$ for $k \geq 3$

Examples:

1. **Petersen Graph**:
   - 10 vertices, 3-regular
   - χ = 3 (not bipartite, has odd cycles)
   - χ' = 4 (Class 2 graph)
2. **Complete Bipartite** $K_{m,n}$:
   - χ = 2 (by definition)
   - χ' = max(m, n)
3. **Cycle** C_7 (odd cycle):
   - χ = 3 (cannot use 2 colors)
   - Need at least 3 due to odd length
4. **Wheel** W_6 (6 outer vertices, 1 center):
   - Outer cycle: 6 vertices (even) → 2 colors
   - Center: Connected to all → 3rd color
   - χ = 3

2.4 Graph Connectivity (40) {#24-graph-connectivity-40}

Key Concepts: Connectivity measures how well a graph holds together. Understanding minimum cuts, articulation points, and bridges is essential for network reliability.

Vertex Connectivity κ(G):

- Minimum number of vertices whose removal disconnects graph
- κ(G) = 0 if G is disconnected or has single vertex
- κ(G) = n-1 for complete graph K_n
- κ(G) ≤ δ(G) (minimum degree)
- **k-connected**: κ(G) ≥ k (needs ≥k vertex removals to disconnect)

Edge Connectivity λ(G):

- Minimum number of edges whose removal disconnects graph
- λ(G) = 0 if G is disconnected
- λ(G) ≤ κ(G) ≤ δ(G) (fundamental inequality)
- **k-edge-connected**: λ(G) ≥ k

Articulation Point (Cut Vertex):

Definition: Vertex whose removal increases number of connected components

- Graph without articulation points is **biconnected**
- Leaf nodes are never articulation points
- In tree: All non-leaf nodes are articulation points

Finding Articulation Points (DFS-based, $O(V + E)$):

For each vertex u, maintain:

- **disc[u]**: Discovery time in DFS
- **low[u]**: Minimum of:
  1. disc[u]
  2. disc[v] for back edge (u,v)
  3. low[v] for tree edge (u,v)

Conditions for Articulation Point:

1. **Root of DFS tree**: u is articulation point ⟺ u has ≥2 children
2. **Non-root**: u is articulation point if ∃ child v where low[v] ≥ disc[u]
   - Means v and descendants can't reach ancestor of u without going through u

Algorithm:

```
DFS(u):
    disc[u] = low[u] = time++
    children = 0
    For each neighbor v:
        If v not visited:
            children++
            parent[v] = u
            DFS(v)
            low[u] = min(low[u], low[v])

            If u is root and children ≥ 2:
                u is articulation point
            If u is not root and low[v] ≥ disc[u]:
                u is articulation point
        Else if v ≠ parent[u]:
            low[u] = min(low[u], disc[v])
```

Bridge (Cut Edge):

Definition: Edge whose removal increases number of connected components

- **No bridge in cycle**: Removing any edge still leaves path
- In tree: All edges are bridges
- Graph without bridges is **2-edge-connected**

Finding Bridges (similar DFS):

Edge (u,v) is bridge ⟺ low[v] > disc[u] for tree edge (u→v)

- Strictly greater (vs ≥ for articulation points)
- Back edges are never bridges

Biconnected Components:

Definition: Maximal biconnected subgraphs

- Every vertex in biconnected component has ≥2 vertex-disjoint paths to any other vertex
- Articulation points can belong to multiple biconnected components
- Non-articulation vertices belong to exactly one

Finding Biconnected Components:

Use DFS with stack:

- Push edges onto stack during DFS
- When articulation point or root with multiple children detected:
  - Pop edges until reaching current edge
  - Popped edges form one biconnected component

Block-Cut Tree:

- Represents biconnected structure
- Nodes: Biconnected components + articulation points
- Edges: Connect articulation point to components containing it
- Always a tree
- Applications: Finding paths avoiding articulation points

Menger's Theorem (Fundamental connectivity result):

Vertex version: Minimum number of vertices separating non-adjacent s,t = Maximum number of internally vertex-disjoint s-t paths

Edge version: Minimum number of edges separating s,t = Maximum number of edge-disjoint s-t paths

Proof idea: Max-flow min-cut in appropriate network

Applications:

1. Network reliability: κ(G) is fault tolerance
2. Finding critical infrastructure: Articulation points
3. Network design: Ensure k-connectivity
4. Clustering: Use bridges to find natural divisions

Strongly Connected Components (Directed Graphs):

Definition: Maximal subgraph where every vertex reaches every other

Kosaraju's Algorithm ($O(V + E)$):

1. DFS on G, record finish times
2. Construct G^T (transpose: reverse all edges)
3. DFS on G^T in decreasing finish time order
4. Each DFS tree in step 3 is one SCC

Why it works:

- If C and C' are SCCs and C→C', then max finish time in C > max finish time in C'
- In G^T: C'→C
- Processing by decreasing finish time ensures we don't escape SCC

Tarjan's Algorithm ($O(V + E)$, one DFS):

Maintain:

- disc[u]: Discovery time
- low[u]: Lowest discovery time reachable
- Stack of vertices in current SCC candidate

Vertex u is root of SCC ⟺ low[u] = disc[u]

Condensation Graph:

- Collapse each SCC to single vertex
- Result is always a DAG
- Used for hierarchical analysis

Problem-Solving Tips:

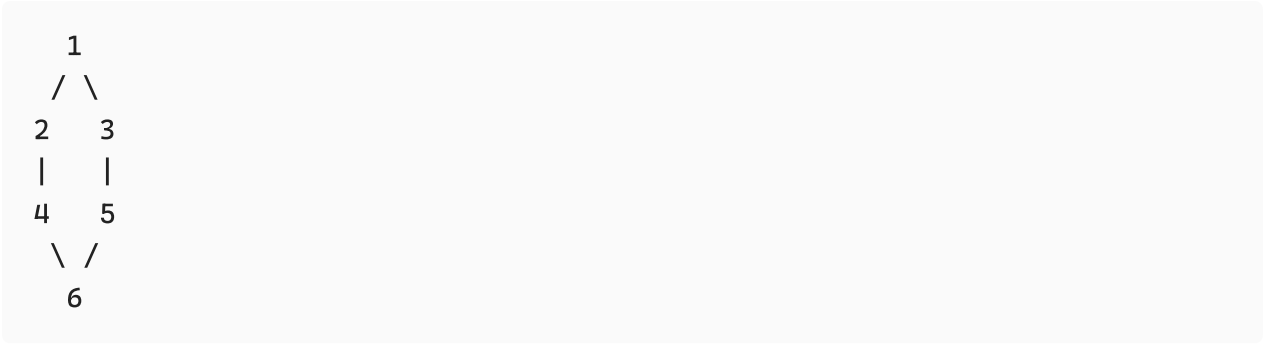
- For articulation points: Use DFS with low values
- Root of DFS tree: Special case (≥2 children)
- Bridges: Stricter condition (low[v] > disc[u])
- For directed graphs: Use Kosaraju or Tarjan for SCCs
- κ(G) ≤ λ(G) ≤ δ(G): Chain of inequalities

GATE Tips:

- Articulation point removal increases components
- Bridge removal increases components
- Complete graph: κ = λ = n-1
- Tree: All non-leaf vertices are articulation points, all edges are bridges
- Biconnected ⟺ No articulation points
- 2-edge-connected ⟺ No bridges
- Menger's theorem: Connectivity = max disjoint paths

Examples:

1. **Path graph** P_4: A-B-C-D
   - Articulation points: B, C
   - Bridges: All 3 edges
   - κ = λ = 1
2. **Cycle** C_4: A-B-C-D-A
   - No articulation points (biconnected)
   - No bridges (2-edge-connected)
   - κ = λ = 2
3. **Complete graph** K_5:
   - κ = λ = 4 (need to remove 4 vertices/edges)
   - No articulation points or bridges
4. **Finding articulation points in graph**:



   - DFS from 1: disc = {1:1, 2:2, 4:3, 6:4, 5:5, 3:6}
   - Articulation points: 4, 6 (removing either disconnects graph)

2.5 Graph Isomorphism (3) {#25-graph-isomorphism-3}

Key Concepts: Two graphs are isomorphic if they are structurally identical, differing only in the names (labels) of their vertices. Determining isomorphism is a fundamental problem in computational complexity, with deep theoretical and practical implications.

Definition: Graphs G = (V, E) and H = (V', E') are **isomorphic** (G ≅ H) if there exists a bijection (a one-to-one correspondence) $f : V \to V'$ such that for any two vertices u, v ∈ V, the edge (u,v) is in E if and only if the edge (f(u), f(v)) is in E'.

- The function f is called an **isomorphism**. It is a structure-preserving map.
- An **automorphism** is an isomorphism from a graph to itself ($f : V \to V$).

Graph Invariants (Properties preserved under isomorphism):

To prove two graphs are *not* isomorphic, it is sufficient to find one structural property (an invariant) that they do not share.

If G ≅ H, then they must have the same:

1. **Number of vertices**: |V| = |V'|
2. **Number of edges**: |E| = |E'|

3. **Degree sequence**: The sorted list of vertex degrees must be identical.
4. **Number of connected components**.
5. **Cycle properties**: Same girth (length of shortest cycle) and same number of cycles of any given length (e.g., triangles, 4-cycles).
6. **Distance properties**: Same diameter (longest shortest path) and radius.
7. **Spectrum**: The eigenvalues of their adjacency matrices are the same.
8. **Chromatic number**: χ(G) = χ(H).
9. **Number of spanning trees**.
10. **Clique number** ω(G) and **Independence number** α(G).

Necessary but Not Sufficient: Sharing all known invariants does not guarantee isomorphism.

- **Classic Counterexample**: A 6-cycle (C_6) and two disjoint 3-cycles ($2 \times K_3$) are not isomorphic, but both have 6 vertices, 6 edges, and are 2-regular (degree sequence (2,2,2,2,2,2)).

Computational Complexity:

The **Graph Isomorphism (GI) problem** asks whether two given graphs G and H are isomorphic. Its complexity is a major open question.

- **GI is in NP**: If two graphs are isomorphic, an isomorphism f serves as a short certificate. We can verify in polynomial time that f is a bijection and preserves all edge relationships.
- **GI is not known to be in P**: No polynomial-time algorithm is known for the general case.
- **GI is not believed to be NP-complete**: If GI were NP-complete, it would cause the **polynomial hierarchy** to collapse to its second level, a result considered unlikely by most complexity theorists.

This places GI in a rare class of problems known as **NP-intermediate**.

- **Breakthrough**: In 2015, László Babai announced a **quasi-polynomial time algorithm** for GI, with a running time of $n^{O((\log n)^c)}$ for some constant c. This is a major theoretical result, suggesting GI is closer to P than to NP-complete problems like SAT.

Algorithmic Approaches:

1. Canonical Labeling:

This is the most powerful practical approach. A **canonical form** is a unique string or graph representation for an isomorphism class.

- A **canonical labeling function** $C(G)$ maps a graph G to its canonical form.
- **Property**: G ≅ H if and only if $C(G) = C(H)$.
- This transforms the isomorphism problem into a string comparison problem.

- **Nauty (No AUTomorphisms, Yes?)** and **Traces** are highly efficient, state-of-the-art tools that compute canonical labelings, solving GI very quickly for most practical graphs, despite having exponential worst-case complexity.

2. Weisfeiler-Leman (WL) Test / Color Refinement:

- An iterative algorithm that partitions vertices based on their own "color" and the multiset of their neighbors' colors.
- If the final color partitions of two graphs differ, they are not isomorphic.
- The 1-dimensional WL test is very fast but fails on some non-isomorphic graphs (e.g., some strongly regular graphs). Higher-dimensional k-WL tests are more powerful but more complex.

Special Cases with Polynomial-Time Solutions:

While the general problem is hard, GI is efficiently solvable for many important classes of graphs:

- **Trees**: Solvable in $O(n)$ time using a recursive canonical labeling approach rooted at the tree's center or bicenter.
- **Planar Graphs**: Solvable in linear time.
- **Graphs of Bounded Degree**: Polynomial time.
- **Graphs of Bounded Genus**: Polynomial time.
- **Graphs of Bounded Treewidth**.
- **Interval Graphs** and **Permutation Graphs**.

However, the problem remains GI-complete (as hard as the general problem) for classes like **regular graphs**, **bipartite graphs**, and **directed acyclic graphs (DAGs)**.

Problem-Solving Strategy:

1. **Invariant Check (Quick Rejection)**: Always start by comparing |V|, |E|, and the degree sequence. This is the fastest way to prove non-isomorphism.
2. **Structural Analysis**: Look for differences in cycle lengths (especially triangles), connectivity (articulation points, bridges), or the presence of specific subgraphs.
3. **Local Neighborhoods**: Compare the neighborhoods of corresponding vertices. For example, if vertex `u` in G has degree 3, and its neighbors have degrees {2, 3, 3}, then its corresponding vertex `v` in H must also have a neighborhood with degrees {2, 3, 3}.
4. **For Small Graphs**: Attempt to construct an isomorphism by hand. Map a vertex in G to a vertex of the same degree in H and extend the mapping outwards, backtracking if a contradiction is found.

GATE Tips:

- **Degree sequence is not enough!** This is a very common trap.

- To prove G ≅ H, you must define the bijection f.
- To prove G ≇ H, you only need to find one invariant that differs.
- Isomorphism for K_n, C_n, P_n is trivial (they are only isomorphic to graphs of the same type and size).
- $K_{m,n}$ is isomorphic to $K_{n,m}$.

2.6 Graph Matching (1) {#26-graph-matching-1}

Key Concepts: A matching is a set of edges with no common vertices. Maximum matching is the largest such set. Critical for assignment problems and network flows.

Matching Definitions:

Matching M: Set of edges where no two edges share a vertex

- Each vertex is incident to at most one edge in M
- **Saturated vertex**: Vertex incident to an edge in M
- **Unsaturated vertex**: Vertex not incident to any edge in M
- **Maximum matching**: Matching with maximum cardinality
- **Maximal matching**: Cannot add more edges (but may not be maximum)
- **Perfect matching**: Every vertex is saturated (size = |V|/2)

Matching Size Bounds:

- Maximum matching size ≤ |V|/2
- For bipartite graph G(X,Y): matching size ≤ min(|X|, |Y|)

Augmenting Paths:

Definition: Path that alternates between non-matching and matching edges

- Starts and ends at unsaturated vertices
- Has odd length
- Endpoints are free (unmatched)

Berge's Theorem (Characterization of maximum matching):

A matching M is maximum ⟺ No augmenting path exists with respect to M

Proof:

- **Forward**: If augmenting path P exists, can increase M by symmetric difference M ⊕ P → M not maximum
- **Backward**: If M not maximum, ∃ larger matching M'. Consider M ⊕ M' (symmetric difference)
  - Forms paths and cycles
  - Cycles have equal matching/non-matching edges

- At least one path has more M' edges than M edges (since |M'| > |M|)
- This path is augmenting path for M

Bipartite Matching:

Bipartite graph G = (X ∪ Y, E) with parts X and Y

Hall's Marriage Theorem (Necessary and sufficient condition):

Bipartite graph G(X,Y) has matching that saturates X ⇔ For every subset S ⊆ X:

$|N(S)| \geq |S|$

where N(S) = neighborhood of S = {y ∈ Y : ∃x ∈ S, (x,y) ∈ E}

Proof (Hall's Theorem):

Necessity (⟹):

- If matching M saturates X, then for any S ⊆ X
- M matches S to |S| distinct vertices in Y
- These |S| vertices are all in N(S)
- Therefore |N(S)| ≥ |S|

Sufficiency (⟸): By induction on |X|

- Base case |X| = 1: Trivial
- **Case 1**: Hall's condition holds with strict inequality for all ∅ ≠ S ⊂ X
  - Pick any edge (x,y), remove x and y
  - Hall's condition still holds for remaining graph
  - By induction, matching exists for remaining graph
  - Add (x,y) to get matching saturating X
- **Case 2**: ∃ S ⊂ X where |N(S)| = |S|
  - By induction, matching M_1 saturates S in G[S ∪ N(S)]
  - Check Hall's condition for XS in remaining graph G(S ∪ N(S))
  - For any T ⊆ XS: $N_{G \setminus (S \cup N(S))}(T) \geq |T|$ (by original Hall's condition)
  - By induction, matching M_2 saturates XS
  - $M_1 \cup M_2$ saturates X

Corollaries of Hall's Theorem:

1. **König's Theorem**: In bipartite graph:
   - Maximum matching size = Minimum vertex cover size
   - Vertex cover: Set of vertices touching all edges
   - Max matching ≤ min vertex cover (always)
   - Equality for bipartite graphs!

2. **Perfect Matching**: Bipartite G(X,Y) with |X| = |Y| has perfect matching ⟺ Hall's condition + regular or near-regular

Algorithms for Maximum Matching:

1. Augmenting Path Algorithm (General approach):

```
MaximumMatching(G):
    M = ∅
    While augmenting path P exists:
        M = M ⊕ P  // Symmetric difference
    Return M
```

2. Hungarian Algorithm / Kuhn's Algorithm (Bipartite):

Time: $O(VE)$

- Use BFS/DFS to find augmenting paths
- Each augmentation increases matching size by 1
- At most |V|/2 augmentations

3. Hopcroft-Karp Algorithm (Bipartite, optimal):

Time: $O(E\sqrt{V})$

- Find maximal set of shortest augmenting paths
- Use BFS to partition vertices by distance
- Use DFS to find disjoint augmenting paths
- At most $O(\sqrt{V})$ phases

4. Blossom Algorithm (Edmo nds, General graphs):

Time: $O(V^2E)$ or $O(V^3)$ with better implementation

- Handle odd cycles (blossoms) by contracting
- Most complex matching algorithm

5. Maximum Weight Matching:

- Assignment problem in bipartite graphs
- Hungarian algorithm: $O(V^3)$
- Finds matching with maximum total weight

König's Theorem (Bipartite graphs):

For bipartite graph G:

$$\text{Maximum Matching Size} = \text{Minimum Vertex Cover Size}$$

Proof sketch:

- Maximum matching ≤ minimum vertex cover (any vertex cover must cover all matching edges)
- Construct vertex cover of size = matching size using Hall's theorem

Applications:

1. **Job assignment**: Workers to jobs (bipartite matching)
2. **Stable marriage problem**: Men and women with preferences
3. **Network flows**: Maximum matching via max-flow
4. **Scheduling**: Non-conflicting time slots
5. **Resource allocation**: Assign resources to tasks
6. **Timetabling**: Course scheduling in universities

Related Concepts:

1. Vertex Cover:

- Set C ⊆ V where every edge has at least one endpoint in C
- In bipartite graphs: min vertex cover = max matching (König)

2. Independent Set:

- Set I ⊆ V with no edges between vertices in I
- Complement of vertex cover
- In bipartite: max independent set = |V| - max matching

3. Edge Cover:

- Set of edges covering all vertices
- Only defined for graphs with no isolated vertices
- Min edge cover = |V| - max matching size

4. Matching Number α'(G):

- Size of maximum matching
- α'(G) ≤ |V|/2

5. Edge Chromatic Number χ'(G):

- Related to matching (edge coloring is partition into matchings)
- χ'(G) = Δ(G) or Δ(G)+1 (Vizing's theorem)

Matching in Specific Graphs:

Complete Graph K_n:

- If n even: Perfect matching exists, size n/2
- If n odd: Maximum matching size (n-1)/2
- Number of perfect matchings (n even): $(n-1)!! = (n-1)(n-3)\cdots 3 \cdot 1$

Complete Bipartite $K_{m,n}$:

- Maximum matching size = min(m,n)
- Perfect matching exists ⟺ m = n

Cycle C_n:

- If n even: Perfect matching, size n/2
- If n odd: Maximum matching size (n-1)/2

Tree:

- Maximum matching found greedily
- Perfect matching rare (requires specific structure)

M-Alternating and M-Augmenting Paths:

M-alternating path: Edges alternate between in M and not in M

M-augmenting path: M-alternating path with both endpoints free

- Length is odd
- Augmenting along this path increases matching size by 1

Tutte's Theorem (General graphs, characterization of perfect matching):

Graph G has perfect matching ⟺ For every subset S ⊆ V:

$o(G-S) \leq |S|$

where o(G-S) = number of odd-sized components in G-S

Tutte-Berge Formula (Maximum matching size):

$\alpha'(G) = \frac{1}{2}\min_{S \subseteq V}(|V| + |S| - o(G-S))$

Problem-Solving Strategy:

1. **Check if bipartite**: Use 2-coloring (BFS/DFS)
2. **For bipartite**: Use Hopcroft-Karp or flow algorithms
3. **For general graphs**: Use Blossom algorithm (complex)
4. **Hall's condition**: Check neighborhoods for bipartite
5. **Perfect matching**: Check if |V| even first

Network Flow Connection:

Maximum matching in bipartite G(X,Y):

- Create source s connected to all x ∈ X (capacity 1)
- Create sink t connected from all y ∈ Y (capacity 1)
- Orient edges X → Y (capacity 1)
- Max flow = max matching size
- Time: $O(VE)$ with Ford-Fulkerson

GATE Tips:

- Hall's theorem: |N(S)| ≥ |S| for all S ⊆ X
- König: max matching = min vertex cover (bipartite only)
- Hopcroft-Karp: $O(E\sqrt{V})$ for bipartite
- Perfect matching requires |V| even
- Augmenting path increases matching by 1
- Berge: No augmenting path ⟺ maximum matching
- For K_{m,n}: max matching = min(m,n)

Examples:

1. **Complete Bipartite** $K_{3,3}$:
   - Perfect matching exists (3 = 3)
   - Size = 3
   - Number of perfect matchings = 3! = 6
2. **Hall's Condition Check**:
   - G(X,Y) where X = {a,b,c}, Y = {1,2,3}
   - Edges: (a,1), (a,2), (b,2), (c,3)
   - Check S = {a,b}: N(S) = {1,2}, |N(S)| = 2 ≥ 2 ✓
   - Check S = {a,b,c}: N(S) = {1,2,3}, |N(S)| = 3 ≥ 3 ✓
   - All subsets satisfy Hall's condition → Perfect matching exists
3. **Augmenting Path Example**:
   - Current matching M = {(a,1), (c,3)}
   - Free vertices: b ∈ X, 2 ∈ Y
   - Augmenting path: b - 2
   - New matching: M' = {(a,1), (b,2), (c,3)}
   - Size increased from 2 to 3
4. **König's Theorem Application**:
   - Bipartite graph with max matching size 4
   - By König: min vertex cover size = 4
   - Can find vertex cover by taking saturated vertices strategically

2.7 Graph Planarity (13) {#27-graph-planarity-13}

Key Concepts: A graph is planar if it can be drawn in the plane with no edge crossings. Fundamental for VLSI design, circuit boards, and geographic networks.

Planar Graph Definition:

Planar graph: Can be embedded in plane such that edges intersect only at vertices

- **Plane graph**: Specific planar embedding
- **Face**: Region bounded by edges (including outer infinite face)
- **Degree of face**: Number of edges bounding it (edges counted twice if border same face both sides)

Euler's Formula (Fundamental relation):

For connected planar graph G with V vertices, E edges, F faces:

$V - E + F = 2$

Proof (Induction on faces):

- Base case: Tree (F = 1): V - (V-1) + 1 = 2 ✓
- Inductive step: Remove edge from cycle
  - Merges 2 faces into 1
  - V unchanged, E decreases by 1, F decreases by 1
  - Formula preserved: V - (E-1) + (F-1) = V - E + F = 2 ✓

Consequences of Euler's Formula:

1. Edge Bound for Simple Planar Graphs:

For simple connected planar graph with V ≥ 3:

$E \leq 3V - 6$

Proof:

- Each face bounded by ≥ 3 edges
- Each edge bounds ≤ 2 faces
- Sum of face degrees: $\sum_f \deg(f) = 2E$ (each edge counted twice)
- Since each face has degree ≥ 3: $2E = \sum \deg(f) \geq 3F$
- Therefore: $F \leq \frac{2E}{3}$
- From Euler: $F = 2 - V + E$
- Substitute: $2 - V + E \leq \frac{2E}{3}$
- Multiply by 3: $6 - 3V + 3E \leq 2E$
- Rearrange: $E \leq 3V - 6$ ✓

2. Edge Bound for Simple Planar Graphs Without Triangles:

If no cycles of length 3 (triangle-free):

$E \leq 2V - 4$

Proof: Similar, but each face has degree ≥ 4

- $2E \geq 4F \rightarrow F \leq E/2$
- Euler: $E/2 \geq F = 2 - V + E$
- Solve: $E \leq 2V - 4$

3. Minimum Degree Bound:

Every planar graph has vertex with degree ≤ 5

Proof (by contradiction):

- Suppose all vertices have degree ≥ 6
- By handshaking: $2E = \sum \deg(v) \geq 6V$
- Therefore: $E \geq 3V$
- But planar: $E \leq 3V - 6$
- Contradiction! ✗

Non-Planar Graphs:

Kuratowski's Theorem (Characterization of non-planarity):

A graph is non-planar ⟺ It contains a subgraph that is a subdivision of K_5 or $K_{3,3}$

Subdivision: Graph obtained by replacing edges with paths

- Also called **homeomorphic** to K_5 or $K_{3,3}$

Wagner's Theorem (Alternative characterization):

A graph is non-planar ⟺ It contains K_5 or $K_{3,3}$ as a **minor**

Minor: Graph obtained by:

1. Deleting edges
2. Deleting vertices
3. Contracting edges (merge endpoints)

Why K_5 and $K_{3,3}$ Non-Planar:

1. K_5 (Complete graph on 5 vertices):

- V = 5, E = 10
- If planar: E ≤ 3V - 6 = 15 - 6 = 9
- But E = 10 > 9

- Violates bound → Non-planar ✗

2. $K_{3,3}$ (Complete bipartite):

- V = 6, E = 9
- Basic bound: E ≤ 3V - 6 = 12 (satisfied)
- But no triangles! Use stronger bound: E ≤ 2V - 4 = 8
- But E = 9 > 8
- Violates bound → Non-planar ✗

Testing Planarity:

1. Euler's Formula Check (Necessary but not sufficient):

- Compute V, E
- Check if E ≤ 3V - 6
- If violated → Non-planar
- If satisfied → May or may not be planar

2. Kuratowski's Theorem (Sufficient and necessary):

- Search for subdivision of K_5 or $K_{3,3}$
- Hard to do by hand for large graphs

3. Left-Right Planarity Test (Efficient algorithm):

- Hopcroft-Tarjan algorithm: $O(V)$
- Lempel-Even-Cederbaum: $O(V)$
- Boyer-Myrvold: $O(V)$ (practical implementation)
- Use DFS and maintain embedding constraints

Planar Graph Properties:

1. Chromatic Number:

- **Four Color Theorem**: χ(G) ≤ 4 for any planar G
- **Five Color Theorem**: χ(G) ≤ 5 (easier proof)
- Many planar graphs need only 3 colors

2. Dual Graph:

- For plane graph G, construct dual G*:
  - Vertex in G* for each face in G
  - Edge in G* for each edge in G (connects adjacent faces)
- Properties:
  - (G*)* = G

- Planar if G planar
- | | V(G*) |
- | | E(G*) |
- | | F(G*) |

3. Outerplanar Graphs:

- Can be drawn with all vertices on outer face boundary
- χ(G) ≤ 3 for outerplanar graphs
- E ≤ 2V - 3 for outerplanar
- Example: Trees are outerplanar

Maximal Planar Graphs:

Maximal planar: Cannot add any edge while maintaining planarity

- Also called **triangulated planar graph**
- Every face (including outer) is triangle
- For V ≥ 3: E = 3V - 6 (achieves bound with equality)
- Examples: K_4, wheel graphs

Euler's Formula Generalizations:

1. Disconnected Graph:

For planar graph with k connected components:

$V - E + F = k + 1$

2. Graph on Surfaces (Genus g):

For graph embedded on surface of genus g:

$V - E + F = 2 - 2g$

where genus g = number of "handles" on surface

- Plane (sphere): g = 0 → V - E + F = 2
- Torus: g = 1 → V - E + F = 0
- K_5 embeds on torus (g = 1)
- $K_{3,3}$ embeds on torus (g = 1)

Planar Graphs in Practice:

1. VLSI Design:

- Circuit layouts on chip
- Minimize crossings (vias cost area)
- Planar preferred for manufacturability

2. Geographic Networks:

- Road networks
- River systems
- Naturally planar due to physical constraints

3. Graph Drawing:

- Visualization algorithms
- Planar drawings more readable
- Straight-line embedding: Fáry's theorem (every planar graph has straight-line embedding)

Fáry's Theorem (Straight-line representability):

Every planar graph can be drawn in plane with:

- Vertices as points
- Edges as straight line segments
- No crossings

Platonic Solids and Planarity:

The 5 Platonic solids correspond to planar graphs:

1. **Tetrahedron**: K_4 (V=4, E=6, F=4)
2. **Cube**: V=8, E=12, F=6
3. **Octahedron**: V=6, E=12, F=8 (dual of cube)
4. **Dodecahedron**: V=20, E=30, F=12
5. **Icosahedron**: V=12, E=30, F=20 (dual of dodecahedron)

All satisfy Euler's formula: V - E + F = 2 ✓

Problem-Solving Strategy:

1. **Small graphs**: Try drawing without crossings
2. **Check bounds**: E ≤ 3V - 6 (necessary condition)
3. **Look for K_5 or $K_{3,3}$ subdivisions**: If found → non-planar
4. **Use algorithms**: For large graphs, implement planarity testing
5. **Special cases**:

- Trees: Always planar
- K_n: Planar ⇔ n ≤ 4
- K_{m,n}: Planar ⇔ m ≤ 2 or n ≤ 2

GATE Tips:

- Euler: V - E + F = 2 (most important formula)
- Edge bound: E ≤ 3V - 6 for simple planar
- K_5 and $K_{3,3}$ are minimal non-planar graphs
- Every planar graph has vertex with degree ≤ 5
- Four color theorem: χ(G) ≤ 4 for planar G
- Maximum edges in planar: 3V - 6
- Outerplanar: E ≤ 2V - 3
- Check edge bounds first before trying to draw

Common Planar Graphs:

- **Trees**: Always planar (E = V - 1, F = 1)
- K_4: Planar (complete graph on 4 vertices)
- $K_{2,n}$: Planar (star graphs)
- **Cycles**: Always planar
- **Wheels** W_n: Planar (cycle with center vertex)
- **Grid graphs**: Planar
- **Petersen graph**: Non-planar (contains $K_{3,3}$ subdivision)

Common Non-Planar Graphs:

- K_5: Complete on 5 vertices (10 edges > 9 bound)
- $K_{3,3}$: Complete bipartite (no triangles, 9 edges > 8 bound)
- K_n **for n ≥ 5**: All non-planar
- $K_{m,n}$ **for m,n ≥ 3**: All non-planar
- **Petersen graph**: Non-planar

Examples:

1. **Is K_4 planar?**
   - V = 4, E = 6
   - Check: E ≤ 3V - 6 = 6 ✓ (with equality)
   - Can draw: Triangle with center point
   - Planar ✓ (also maximal planar)
2. **Prove $K_{3,3}$ non-planar**:
   - V = 6, E = 9
   - No triangles (bipartite)

- Bound: E ≤ 2V - 4 = 8
- But E = 9 > 8 ✗
- Non-planar

3. **Euler's formula verification** on cube:
   - V = 8 (corners)
   - E = 12 (edges)
   - F = 6 (faces)
   - V - E + F = 8 - 12 + 6 = 2 ✓
4. **Maximum edges** in planar graph on 10 vertices:
   - E ≤ 3V - 6 = 3(10) - 6 = 24
   - Maximum: 24 edges

3. Discrete Mathematics: Mathematical Logic (77)

3.1 First Order Logic (34) {#31-first-order-logic-34}

Key Concepts: First-Order Logic (FOL), also called Predicate Logic or First-Order Predicate Calculus, extends propositional logic with quantifiers and predicates to express statements about objects and their relationships.

Fundamental Components:

1. Vocabulary (Signature):

- **Constants**: Specific objects (a, b, c, 0, 1, ...)
- **Variables**: Placeholders for objects (x, y, z, ...)
- **Function symbols**: Maps objects to objects (f(x), g(x,y), succ(n), ...)
- **Predicate symbols**: Relations that return true/false (P(x), Q(x,y), Loves(x,y), ...)
- **Arity**: Number of arguments a function/predicate takes

2. Terms:

- Every constant is a term
- Every variable is a term
- If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term
- Example: $f(x, g(a, y))$ is a term

3. Atomic Formulas:

- If P is an n-ary predicate and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atomic formula
- Equality: $t_1 = t_2$ is atomic (in FOL with equality)

4. Well-Formed Formulas (WFFs):

- Every atomic formula is a WFF

- If ϕ is a WFF, then $\neg\phi$ is a WFF
- If ϕ, ψ are WFFs, then $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \to \psi)$, $(\phi \leftrightarrow \psi)$ are WFFs
- If ϕ is a WFF and x is a variable, then $\forall x\phi$ and $\exists x\phi$ are WFFs

Quantifiers:

Universal Quantifier ($\forall$):

- $\forall x P(x)$: "For all x, P(x) holds"
- True iff P(x) is true for every element in the domain
- Semantic: $\forall x P(x) \equiv P(a_1) \wedge P(a_2) \wedge \cdots$ (conjunction over domain)

Existential Quantifier ($\exists$):

- $\exists x P(x)$: "There exists an x such that P(x) holds"
- True iff P(x) is true for at least one element in the domain
- Semantic: $\exists x P(x) \equiv P(a_1) \vee P(a_2) \vee \cdots$ (disjunction over domain)

Uniqueness Quantifier ($\exists!$):

- $\exists! x P(x)$: "There exists exactly one x such that P(x)"
- Definition: $\exists! x P(x) \equiv \exists x(P(x) \wedge \forall y(P(y) \to y = x))$

Formulas/Theorems:

Quantifier Negation Laws (De Morgan for Quantifiers):

$$\neg\forall x P(x) \equiv \exists x \neg P(x)$$

$$\neg\exists x P(x) \equiv \forall x \neg P(x)$$

Proof:

- "Not all are P" means "at least one is not P"
- "None exists with P" means "all are not P"

Quantifier Distribution:

$$\forall x(P(x) \wedge Q(x)) \equiv (\forall x P(x)) \wedge (\forall x Q(x))$$

$$\exists x(P(x) \vee Q(x)) \equiv (\exists x P(x)) \vee (\exists x Q(x))$$

Warning - Non-equivalences (Common GATE traps):

$$\forall x(P(x) \vee Q(x)) \not\equiv (\forall x P(x)) \vee (\forall x Q(x))$$

$$\exists x(P(x) \wedge Q(x)) \not\equiv (\exists x P(x)) \wedge (\exists x Q(x))$$

Counterexample: Let domain = {1, 2}, P(1)=T, P(2)=F, Q(1)=F, Q(2)=T

- LHS: $\forall x(P(x) \vee Q(x)) = (T \vee F) \wedge (F \vee T) = T$

- RHS: $(\forall x P(x)) \vee (\forall x Q(x)) = F \vee F = F$

Quantifier Order:

$$\forall x \forall y P(x, y) \equiv \forall y \forall x P(x, y)$$

$$\exists x \exists y P(x, y) \equiv \exists y \exists x P(x, y)$$

$$\forall x \exists y P(x, y) \not\equiv \exists y \forall x P(x, y)$$

Critical difference:

- $\forall x \exists y P(x, y)$: "For each x, there is some y (possibly different for each x)"
- $\exists y \forall x P(x, y)$: "There is one y that works for all x"
- The second is strictly stronger: $\exists y \forall x P(x, y) \Rightarrow \forall x \exists y P(x, y)$

Free and Bound Variables:

- **Bound variable**: Appears within scope of a quantifier
- **Free variable**: Not bound by any quantifier
- **Closed formula (Sentence)**: No free variables

Example: In $\forall x (P(x, y) \to Q(x))$

- x is bound (by $\forall x$)
- y is free

Substitution:

$\phi[t/x]$ denotes replacing all free occurrences of x with term t

Capture-avoiding substitution: t must not contain variables that would become bound

Prenex Normal Form (PNF):

A formula is in PNF if all quantifiers are at the front:

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi$$

where each $Q_i \in \{\forall, \exists\}$ and ϕ is quantifier-free (the **matrix**)

Conversion to PNF:

1. Eliminate $\to$ and $\leftrightarrow$:
   - $P \to Q \equiv \neg P \vee Q$
   - $P \leftrightarrow Q \equiv (P \to Q) \wedge (Q \to P)$
2. Push negations inward using:
   - $\neg\neg P \equiv P$
   - $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
   - $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

- $\neg \forall x P \equiv \exists x \neg P$
- $\neg \exists x P \equiv \forall x \neg P$

3. Rename bound variables to avoid conflicts
4. Move quantifiers to front using:
   - $(Qx P(x)) \vee R \equiv Qx(P(x) \vee R)$ (if x not free in R)
   - $(Qx P(x)) \wedge R \equiv Qx(P(x) \wedge R)$ (if x not free in R)

Example:

Convert $\neg \forall x(P(x) \rightarrow \exists y Q(x, y))$ to PNF:

1. $\neg \forall x(\neg P(x) \vee \exists y Q(x, y))$
2. $\exists x \neg(\neg P(x) \vee \exists y Q(x, y))$
3. $\exists x(P(x) \wedge \neg \exists y Q(x, y))$
4. $\exists x(P(x) \wedge \forall y \neg Q(x, y))$
5. $\exists x \forall y(P(x) \wedge \neg Q(x, y))$

Skolemization:

Process to eliminate existential quantifiers (for resolution theorem proving):

Rules:

1. Convert to PNF first
2. For $\exists y$ not preceded by any $\forall$: Replace y with new constant c (Skolem constant)
3. For $\exists y$ preceded by $\forall x_1 \ldots \forall x_n$: Replace y with $f(x_1, \ldots, x_n)$ (Skolem function)

Example:

$\forall x \exists y \forall z \exists w P(x, y, z, w)$

1. y depends on x: Replace with $f(x)$
2. w depends on x and z: Replace with $g(x, z)$
3. Result: $\forall x \forall z P(x, f(x), z, g(x, z))$

Important: Skolemization preserves satisfiability but NOT equivalence

Semantic Concepts:

Interpretation (Structure):

An interpretation $\mathcal{I}$ consists of:

- **Domain** D: Non-empty set of objects
- **Mapping** for each constant to an element of D
- **Mapping** for each function symbol to a function on D
- **Mapping** for each predicate symbol to a relation on D

Satisfiability and Validity:

- **Satisfiable**: True in at least one interpretation
- **Valid (Tautology)**: True in ALL interpretations
- **Unsatisfiable (Contradiction)**: False in ALL interpretations

Logical Consequence:

$\Gamma \models \phi$: In every interpretation where all formulas in Γ are true, ϕ is also true

Important Metatheorems:

Compactness Theorem: If every finite subset of Γ is satisfiable, then Γ is satisfiable

Löwenheim-Skolem Theorem: If a FOL theory has an infinite model, it has models of all infinite cardinalities

Gödel's Completeness Theorem: $\Gamma \models \phi$ iff $\Gamma \vdash \phi$ (semantic entailment = syntactic provability)

Decidability: FOL is semi-decidable (can enumerate theorems) but undecidable in general (Church-Turing)

Problem-Solving Tips:

1. Translating English to FOL:

- "All S are P": $\forall x(S(x) \rightarrow P(x))$ (NOT $\forall x(S(x) \land P(x))$!)
- "Some S are P": $\exists x(S(x) \land P(x))$ (NOT $\exists x(S(x) \rightarrow P(x))$!)
- "No S are P": $\neg\exists x(S(x) \land P(x))$ OR $\forall x(S(x) \rightarrow \neg P(x))$
- "Only S are P": $\forall x(P(x) \rightarrow S(x))$

GATE Trap: "All" uses implication; "Some" uses conjunction

2. Negating Quantified Statements:

Systematically apply: $\neg\forall = \exists\neg$ and $\neg\exists = \forall\neg$

Example: Negate "Everyone loves someone"

- Original: $\forall x \exists y Loves(x, y)$
- Negation: $\neg\forall x \exists y Loves(x, y)$
- = $\exists x \neg\exists y Loves(x, y)$
- = $\exists x \forall y \neg Loves(x, y)$
- English: "Someone loves nobody"

3. Checking Validity:

- Find counterexample (interpretation where formula is false) to disprove validity

- Use resolution or tableaux method to prove validity

4. Determining Satisfiability:

- Construct an interpretation that makes the formula true
- Small domains often suffice for counterexamples

Common GATE Patterns:

1. **Quantifier scope ambiguity**: Carefully parse nested quantifiers
2. **Order of quantifiers**: $\forall\exists$ vs $\exists\forall$ - check if order matters
3. **Domain sensitivity**: Empty domain edge cases (though GATE assumes non-empty)
4. **Translation errors**: "All" with implication, "Some" with conjunction
5. **Negation pushing**: Methodically apply De Morgan + quantifier negation

Examples:

Example 1: Express "Every student who studies passes" in FOL

Let $S(x)$: x is a student, $St(x)$: x studies, $P(x)$: x passes

Answer: $\forall x((S(x) \land St(x)) \rightarrow P(x))$

Example 2: Is $\forall x P(x) \lor \forall x Q(x) \rightarrow \forall x(P(x) \lor Q(x))$ valid?

Yes. Proof: Assume antecedent true.

- Case 1: $\forall x P(x)$ true → For any x, $P(x)$ true → $P(x) \lor Q(x)$ true → $\forall x(P(x) \lor Q(x))$ true
- Case 2: $\forall x Q(x)$ true → Similar reasoning

Example 3: Is $\forall x(P(x) \lor Q(x)) \rightarrow \forall x P(x) \lor \forall x Q(x)$ valid?

No. Counterexample: Domain = {a, b}, P(a)=T, P(b)=F, Q(a)=F, Q(b)=T

- LHS: $\forall x(P(x) \lor Q(x)) = (T \lor F) \land (F \lor T) = T$
- RHS: $\forall x P(x) \lor \forall x Q(x) = F \lor F = F$
- Implication: T → F = F (not valid)

Example 4: Skolemize $\exists x \forall y \exists z(P(x, y) \rightarrow Q(y, z))$

1. x has no preceding universal: Replace with constant c
2. z is preceded by $\forall y$: Replace with $f(y)$
3. Result: $\forall y(P(c, y) \rightarrow Q(y, f(y)))$

Example 5: Express "There is a barber who shaves all those who don't shave themselves"

$\exists x(Barber(x) \land \forall y(\neg Shaves(y, y) \rightarrow Shaves(x, y)))$

(This leads to Russell's Paradox when asking if the barber shaves himself)

3.2 Logical Reasoning (3) {#32-logical-reasoning-3}

Key Concepts: Logical reasoning encompasses the systematic methods for deriving conclusions from premises. It includes valid argument forms, inference rules, and the identification of fallacies.

Types of Reasoning:

1. Deductive Reasoning:

- Conclusion follows necessarily from premises
- If premises are true, conclusion MUST be true
- Goes from general to specific
- **Validity**: Argument form is valid if true premises guarantee true conclusion
- **Soundness**: Valid argument with actually true premises

2. Inductive Reasoning:

- Conclusion is probable given premises
- Goes from specific observations to general conclusions
- Cannot guarantee truth of conclusion
- Example: "All observed swans are white" → "All swans are white" (defeasible)

3. Abductive Reasoning:

- Inference to the best explanation
- Given observation, infer most likely cause
- Example: "Grass is wet" → "It probably rained" (but could be sprinklers)

Valid Argument Forms (Rules of Inference):

Formulas/Theorems:

1. Modus Ponens (MP) - Affirming the Antecedent:

$$\frac{P \to Q, \quad P}{Q}$$

- If P implies Q, and P is true, then Q is true
- Example: "If it rains, ground is wet. It rains. Therefore, ground is wet."

2. Modus Tollens (MT) - Denying the Consequent:

$$\frac{P \to Q, \quad \neg Q}{\neg P}$$

- If P implies Q, and Q is false, then P is false
- Example: "If guilty, evidence exists. No evidence. Therefore, not guilty."

3. Hypothetical Syllogism (HS) - Chain Rule:

$$\frac{P \to Q, \quad Q \to R}{P \to R}$$

- Transitivity of implication
- Example: "If A then B. If B then C. Therefore, if A then C."

4. Disjunctive Syllogism (DS):

$$\frac{P \lor Q, \quad \neg P}{Q}$$

- If one of P or Q is true, and P is false, then Q is true
- Example: "Tea or coffee. Not tea. Therefore, coffee."

5. Constructive Dilemma:

$$\frac{(P \to Q) \land (R \to S), \quad P \lor R}{Q \lor S}$$

6. Destructive Dilemma:

$$\frac{(P \to Q) \land (R \to S), \quad \neg Q \lor \neg S}{\neg P \lor \neg R}$$

7. Conjunction (Conj):

$$\frac{P, \quad Q}{P \land Q}$$

8. Simplification (Simp):

$$\frac{P \land Q}{P}$$

9. Addition (Add):

$$\frac{P}{P \lor Q}$$

10. Resolution:

$$\frac{P \lor Q, \quad \neg P \lor R}{Q \lor R}$$

Categorical Syllogisms:

Classical Aristotelian logic dealing with categories (sets) of objects.

Standard Form:

- Major premise: Statement about relationship between two categories
- Minor premise: Statement relating a third category

- Conclusion: Statement relating first and third categories

Four Categorical Forms:

Form	Name	Statement	Example
A	Universal Affirmative	All S are P	All dogs are mammals
E	Universal Negative	No S are P	No cats are dogs
I	Particular Affirmative	Some S are P	Some birds are pets
O	Particular Negative	Some S are not P	Some animals are not pets

Valid Syllogistic Forms (Examples):

Barbara (AAA-1):

- All M are P
- All S are M
- ∴ All S are P

Celarent (EAE-1):

- No M are P
- All S are M
- ∴ No S are P

Darii (All-1):

- All M are P
- Some S are M
- ∴ Some S are P

Testing Validity with Venn Diagrams:

1. Draw three overlapping circles for S, P, M
2. Diagram premises (shade for "no", X for "some exists")
3. Check if conclusion necessarily follows

Logical Fallacies:

Formal Fallacies (Invalid argument forms):

1. Affirming the Consequent:

$$\frac{P \to Q, \quad Q}{P} \quad \text{INVALID!}$$

- Example: "If it rains, ground is wet. Ground is wet. Therefore, it rained." (Could be sprinklers)

2. Denying the Antecedent:

$$\frac{P \to Q, \quad \neg P}{\neg Q} \quad \text{INVALID!}$$

- Example: "If I study, I pass. I didn't study. Therefore, I won't pass." (Could pass anyway)

3. Undistributed Middle:

- All A are B
- All C are B
- ∴ All A are C (INVALID)
- Example: "All dogs are animals. All cats are animals. Therefore, all dogs are cats."

4. Illicit Major/Minor:

- Conclusion distributes a term not distributed in premises

Informal Fallacies (Content-based errors):

1. Ad Hominem: Attacking the person, not the argument
2. Straw Man: Misrepresenting opponent's argument
3. Appeal to Authority: Using authority instead of evidence
4. False Dilemma: Presenting only two options when more exist
5. Slippery Slope: Claiming one event inevitably leads to extreme consequences
6. Circular Reasoning: Assuming conclusion in premises
7. Hasty Generalization: Drawing broad conclusions from limited examples
8. Post Hoc: Assuming causation from correlation

Problem-Solving Tips:

1. Identify Argument Structure:

- Find conclusion (often after "therefore", "thus", "hence")
- Identify premises (often after "since", "because", "given")
- Determine argument form

2. Test Validity:

- **Truth Table Method**: Construct table, check if premises true and conclusion false is possible
- **Counterexample Method**: Find interpretation where premises true but conclusion false
- **Derivation Method**: Derive conclusion from premises using valid rules

3. Recognize Common Patterns:

- Chain of implications → Hypothetical Syllogism
- Elimination of alternatives → Disjunctive Syllogism
- Conditional with affirmed antecedent → Modus Ponens
- Conditional with denied consequent → Modus Tollens

4. Watch for Fallacies:

- Does conclusion follow necessarily?
- Are there hidden assumptions?
- Is the argument form valid?

GATE-Specific Strategies:

1. **Translate to symbols**: Convert English to logical form
2. **Check validity separately from truth**: Valid arguments can have false premises
3. **Look for standard forms**: MP, MT, HS, DS are most common
4. **Beware of "affirming consequent" and "denying antecedent" traps

Examples:

Example 1: Determine validity
"All programmers are logical. John is a programmer. Therefore, John is logical."

- $\forall x(P(x) \rightarrow L(x)), P(John) \therefore L(John)$
- Form: Universal instantiation + Modus Ponens
- **Valid**

Example 2: Identify the fallacy
"If you're a good student, you get good grades. You got good grades. So you're a good student."

- Form: $P \rightarrow Q, Q \therefore P$
- **Fallacy**: Affirming the Consequent

Example 3: Valid or Invalid?
"Either the server is down or the network is slow. The server is not down. Therefore, the network is slow."

- Form: $P \vee Q, \neg P \therefore Q$
- This is Disjunctive Syllogism
- **Valid**

Example 4: Complete the argument
"If taxes increase, spending decreases. If spending decreases, unemployment rises. Taxes increased. Therefore, ___"

- $T \to S$, $S \to U$, T
- By HS: $T \to U$
- By MP: U
- **Answer**: "Unemployment rises"

Example 5: Truth table verification for Modus Ponens

P	Q	P→Q	(P→Q) ∧ P	((P→Q) ∧ P) → Q
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

The implication $((P \to Q) \land P) \to Q$ is a tautology, confirming MP is valid.

3.3 Propositional Logic (40) {#33-propositional-logic-40}

Key Concepts: Propositional logic (also called sentential logic or Boolean logic) is the study of logical relationships between propositions (statements that are either true or false).

Fundamental Components:

Propositions:

- Declarative statements with definite truth value (T or F)
- Atomic propositions: Cannot be broken down (P, Q, R, ...)
- Compound propositions: Formed using logical connectives

Logical Connectives:

Symbol	Name	Meaning	Alternative Notation
¬	Negation	NOT	~, !, $\overline{P}$
∧	Conjunction	AND	&, ·
∨	Disjunction	OR (inclusive)	|, +
→	Implication	IF...THEN	⊃, ⇒
↔	Biconditional	IF AND ONLY IF	≡, ⇔
⊕	XOR	Exclusive OR	≢
↓	NOR	NOT OR (Peirce arrow)	
↑	NAND	NOT AND (Sheffer stroke)	|

Truth Tables for Basic Connectives:

Negation (¬P):
P	¬P
T	F
F	T

Conjunction (P ∧ Q):
P	Q	P ∧ Q
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction (P ∨ Q) (Inclusive OR):
P	Q	P ∨ Q
T	T	T
T	F	T
F	T	T
F	F	F

Implication (P → Q):
P	Q	P → Q
T	T	T
T	F	F
F	T	T
F	F	T

Key insight: P → Q is false ONLY when P is true and Q is false

Equivalent forms: $P \to Q \equiv \neg P \lor Q \equiv \neg Q \to \neg P$

Biconditional (P ↔ Q):
P	Q	P ↔ Q
T	T	T
T	F	F
F	T	F
F	F	T

Key insight: P ↔ Q is true when P and Q have same truth value

Equivalent: $(P \to Q) \land (Q \to P)$

XOR ($P \oplus Q$):

P	Q	$P \oplus Q$
T	T	F
T	F	T
F	T	T
F	F	F

Equivalent: $(P \vee Q) \wedge \neg(P \wedge Q)$ or $(P \wedge \neg Q) \vee (\neg P \wedge Q)$

Formulas/Theorems:

Logical Equivalences (Laws):

1. Identity Laws:

$$P \wedge T \equiv P$$
$$P \vee F \equiv P$$

2. Domination Laws:

$$P \vee T \equiv T$$
$$P \wedge F \equiv F$$

3. Idempotent Laws:

$$P \vee P \equiv P$$
$$P \wedge P \equiv P$$

4. Double Negation:

$$\neg(\neg P) \equiv P$$

5. Commutative Laws:

$$P \vee Q \equiv Q \vee P$$
$$P \wedge Q \equiv Q \wedge P$$

6. Associative Laws:

$$(P \vee Q) \vee R \equiv P \vee (Q \vee R)$$
$$(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$$

7. Distributive Laws:

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$
$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$

8. De Morgan's Laws:

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$\neg(P \lor Q) \equiv \neg P \land \neg Q$$

Proof of first De Morgan's law (by truth table):

P	Q	P∧Q	¬(P∧Q)	¬P	¬Q	¬P∨¬Q
T	T	T	F	F	F	F
T	F	F	T	F	T	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

Columns 4 and 7 are identical. ■

9. Absorption Laws:

$$P \lor (P \land Q) \equiv P$$

$$P \land (P \lor Q) \equiv P$$

10. Negation Laws:

$$P \lor \neg P \equiv T$$

(Law of Excluded Middle)

$$P \land \neg P \equiv F$$

(Law of Non-Contradiction)

Implication Equivalences:

$$P \to Q \equiv \neg P \lor Q$$

(Material implication)

$$P \to Q \equiv \neg Q \to \neg P$$

(Contrapositive)

$$\neg(P \to Q) \equiv P \land \neg Q$$

$$P \to Q \equiv \neg P \lor Q \equiv \neg(P \land \neg Q)$$

Biconditional Equivalences:

$$P \leftrightarrow Q \equiv (P \to Q) \land (Q \to P)$$

$$P \leftrightarrow Q \equiv (P \land Q) \lor (\neg P \land \neg Q)$$

$$P \leftrightarrow Q \equiv \neg(P \oplus Q)$$

Functional Completeness:

A set of connectives is **functionally complete** if every Boolean function can be expressed using only connectives from that set.

Functionally Complete Sets:

- {¬, ∧}
- {¬, ∨}
- {¬, →}
- {↑} (NAND alone)
- {↓} (NOR alone)

Proof that {NAND} is complete:

- $\neg P \equiv P \uparrow P$
- $P \wedge Q \equiv \neg(P \uparrow Q) \equiv (P \uparrow Q) \uparrow (P \uparrow Q)$
- $P \vee Q \equiv (P \uparrow P) \uparrow (Q \uparrow Q)$

Normal Forms:

Literal: A variable or its negation (P or ¬P)

Clause: Disjunction of literals (for CNF) or conjunction of literals (for DNF)

Conjunctive Normal Form (CNF):

- Conjunction of disjunctions: $(l_1 \vee l_2) \wedge (l_3 \vee l_4) \wedge \ldots$
- Also called Product of Sums (POS)
- Example: $(P \vee Q) \wedge (\neg P \vee R) \wedge (Q \vee \neg R)$

CNF Conversion Algorithm:

1. Eliminate ↔: $P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$
2. Eliminate →: $P \rightarrow Q \equiv \neg P \vee Q$
3. Push ¬ inward using De Morgan and double negation
4. Distribute ∨ over ∧: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

Disjunctive Normal Form (DNF):

- Disjunction of conjunctions: $(l_1 \wedge l_2) \vee (l_3 \wedge l_4) \vee \ldots$
- Also called Sum of Products (SOP)
- Example: $(P \wedge Q) \vee (\neg P \wedge R) \vee (Q \wedge \neg R)$

DNF from Truth Table:

1. Find all rows where formula is TRUE
2. For each such row, create a minterm (conjunction of all literals, positive if T, negative if F)
3. Disjoin all minterms

Example: For $P \rightarrow Q$:

P	Q	P→Q	Minterm
T	T	T	P ∧ Q
T	F	F	-
F	T	T	¬P ∧ Q
F	F	T	¬P ∧ ¬Q

DNF: $(P \wedge Q) \vee (\neg P \wedge Q) \vee (\neg P \wedge \neg Q)$

Semantic Concepts:

Tautology: Formula true under ALL interpretations

- Example: $P \vee \neg P$

Contradiction (Unsatisfiable): Formula false under ALL interpretations

- Example: $P \wedge \neg P$

Contingency: Formula true under SOME interpretations, false under others

- Example: $P \wedge Q$

Satisfiable: Formula true under AT LEAST ONE interpretation

- Tautologies and contingencies are satisfiable
- Contradictions are unsatisfiable

Logical Entailment: $\phi \models \psi$ means whenever ϕ is true, ψ is true

Logical Equivalence: $\phi \equiv \psi$ means $\phi \models \psi$ and $\psi \models \phi$

Resolution:

A complete inference rule for refutation in CNF:

$$\frac{C_1 \vee P, \quad C_2 \vee \neg P}{C_1 \vee C_2}$$

Resolution Refutation (to prove $\Gamma \models \phi$):

1. Convert $\Gamma \cup \{\neg\phi\}$ to CNF
2. Repeatedly apply resolution to derive new clauses
3. If empty clause (□) derived → $\Gamma \models \phi$ (proof by contradiction)
4. If no more clauses derivable without □ → $\Gamma \not\models \phi$

Example: Prove $\{P \to Q, Q \to R\} \models P \to R$

1. Premises in CNF: $\neg P \vee Q$, $\neg Q \vee R$
2. Negated conclusion: $\neg(P \to R) \equiv P \wedge \neg R$
3. Clauses: $\{\neg P \vee Q, \neg Q \vee R, P, \neg R\}$
4. Resolution steps:
   - Resolve $\neg P \vee Q$ with P: Q
   - Resolve $\neg Q \vee R$ with Q: R
   - Resolve R with $\neg R$: $\square$ (empty clause)
5. Empty clause derived → Original entailment holds ■

SAT Problem:

Satisfiability (SAT): Given a propositional formula, is there an assignment making it true?

Complexity: SAT is NP-complete (Cook-Levin theorem)

2-SAT: CNF where each clause has at most 2 literals

- **Polynomial time solvable** (O(n+m) using implication graph + SCC)
- Key: $(P \vee Q) \equiv (\neg P \to Q) \wedge (\neg Q \to P)$

3-SAT: CNF where each clause has exactly 3 literals

- **NP-complete** (even this restricted form)

Horn Clauses:

- Clause with at most one positive literal
- Forms: $\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n \vee Q$ (definite clause)
- Equivalent: $P_1 \wedge P_2 \wedge \ldots \wedge P_n \to Q$
- SAT for Horn clauses is P (unit propagation)

Problem-Solving Tips:

1. Truth Table Method:

- List all 2^n combinations for n variables
- Compute formula value for each
- Useful for: Proving equivalence, finding satisfying assignments, checking tautology

2. Algebraic Manipulation:

- Apply logical equivalences to simplify
- Useful for: Quick equivalence checks, normal form conversion

3. Short-Circuit Evaluation:

- For tautology: Find structure that's always true (e.g., $A \vee \neg A$ within formula)

- For contradiction: Find structure that's always false

4. Implication Truth Table Trick:

- $P \to Q$ is false ONLY when P=T, Q=F
- To disprove $\phi \to \psi$: Find case where ϕ true and ψ false

5. Counting Arguments:

- n variables $\to 2^n$ possible interpretations
- Tautology: True in all 2^n
- Contradiction: True in 0
- Number of satisfying assignments can reveal structure

Common GATE Patterns and Traps:

1. Implication Direction:

- "P is sufficient for Q" $\to P \to Q$
- "P is necessary for Q" $\to Q \to P$
- "P only if Q" $\to P \to Q$ (NOT $Q \to P$!)

2. Negating Implications:

- $\neg(P \to Q) \equiv P \wedge \neg Q$ (NOT $\neg P \vee \neg Q$!)

3. Contrapositive vs Converse:

- Contrapositive ($\neg Q \to \neg P$): Equivalent to $P \to Q$
- Converse ($Q \to P$): NOT equivalent to $P \to Q$
- Inverse ($\neg P \to \neg Q$): NOT equivalent to $P \to Q$

4. Distribution Errors:

- $\neg$ doesn't distribute over $\wedge$ or $\vee$ directly
- Must use De Morgan's laws

5. CNF/DNF Confusion:

- CNF: AND of ORs (clauses connected by ∧)
- DNF: OR of ANDs (terms connected by ∨)

Examples:

Example 1: Simplify $(P \to Q) \wedge (P \to \neg Q)$

$\equiv (\neg P \vee Q) \wedge (\neg P \vee \neg Q)$
$\equiv \neg P \vee (Q \wedge \neg Q)$ (Distribution)

$\equiv \neg P \vee F$
$\equiv \neg P$

Example 2: Is $(P \to Q) \to P) \to P$ a tautology? (Peirce's Law)

Truth table approach:

P	Q	P→Q	(P→Q)→P	((P→Q)→P)→P
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	F	T

All T → **Tautology** ✓

Example 3: Convert to CNF: $P \leftrightarrow (Q \to R)$

1. $P \leftrightarrow (Q \to R)$
2. $(P \to (Q \to R)) \wedge ((Q \to R) \to P)$
3. $(P \to (\neg Q \vee R)) \wedge ((\neg Q \vee R) \to P)$
4. $(\neg P \vee \neg Q \vee R) \wedge (\neg(\neg Q \vee R) \vee P)$
5. $(\neg P \vee \neg Q \vee R) \wedge ((Q \wedge \neg R) \vee P)$
6. $(\neg P \vee \neg Q \vee R) \wedge (P \vee Q) \wedge (P \vee \neg R)$

Example 4: Use resolution to prove $(P \vee Q) \wedge (\neg P \vee R) \wedge (\neg Q \vee R) \models R$

Negate conclusion: $\neg R$

Clauses: $\{P \vee Q, \neg P \vee R, \neg Q \vee R, \neg R\}$

Resolution:

1. $\neg P \vee R$ + $\neg R \to \neg P$
2. $\neg Q \vee R$ + $\neg R \to \neg Q$
3. $P \vee Q$ + $\neg P \to Q$
4. Q + $\neg Q \to \square$

Empty clause derived → entailment holds ✓

Example 5: How many satisfying assignments for $(P \vee Q) \wedge (P \vee \neg Q) \wedge R$?

$(P \vee Q) \wedge (P \vee \neg Q) \equiv P \vee (Q \wedge \neg Q) \equiv P$

So formula simplifies to $P \wedge R$

- P must be T (1 choice)
- R must be T (1 choice)

- Q can be T or F (2 choices)

Total: $1 \times 1 \times 2 = 2$ satisfying assignments

Example 6: Prove $\{P \vee Q, P \to R, Q \to R\} \models R$ using resolution

Clauses from premises:

- $P \vee Q$
- $\neg P \vee R$
- $\neg Q \vee R$

Add negated conclusion: $\neg R$

Resolution:

1. $\neg P \vee R$ + $\neg R \to \neg P$
2. $\neg Q \vee R$ + $\neg R \to \neg Q$
3. $P \vee Q$ + $\neg P \to Q$
4. Q + $\neg Q \to \square$

Proved ✓ (This is called "proof by cases" in natural deduction)

1. Discrete Mathematics: Set Theory & Algebra (171)

1.1 Binary Operation (8) {#11-binary-operation-8}

Key Concepts: A binary operation on a set is a function that combines two elements to produce another element in the same set. Foundation for algebraic structures like groups, rings, and fields.

Definition: A binary operation $*$ on set S is a function $* : S \times S \to S$

Properties of Binary Operations:

1. Closure: $\forall a, b \in S : a * b \in S$

- The result of the operation is always in the original set
- Example: Addition on integers is closed, but division is not (since $1 \div 2 \notin \mathbb{Z}$)

2. Associativity: $\forall a, b, c \in S : (a * b) * c = a * (b * c)$

- Order of operations doesn't matter for grouping
- Example: $(2 + 3) + 4 = 2 + (3 + 4) = 9$
- Counterexample: Subtraction is not associative: $(5 - 3) - 2 \neq 5 - (3 - 2)$

3. Commutativity: $\forall a, b \in S : a * b = b * a$

- Order of operands doesn't matter

- Example: Multiplication is commutative, but matrix multiplication is not

4. Identity Element: $\exists e \in S$ such that $\forall a \in S : a * e = e * a = a$

- Neutral element that doesn't change other elements
- Must be unique if it exists
- **Proof of uniqueness**: If e_1 and e_2 are both identities, then $e_1 = e_1 * e_2 = e_2$

5. Inverse Element: For $a \in S$, element $a^{-1} \in S$ such that $a * a^{-1} = a^{-1} * a = e$

- Only exists if identity exists
- **Uniqueness**: If b and c are both inverses of a, then:
  $b = b * e = b * (a * c) = (b * a) * c = e * c = c$

Algebraic Structures:

Semigroup: $(S, *)$ where $*$ is associative and closed

- Example: $(\mathbb{N}, +)$ - natural numbers under addition

Monoid: Semigroup with identity element

- Example: $(\mathbb{N} \cup \{0\}, +)$ with identity 0
- Example: $(\{0, 1\}^*, \cdot)$ - strings under concatenation with empty string as identity

Group: Monoid where every element has an inverse

- **Group Axioms**:
  1. **Closure**: $\forall a, b \in G : a * b \in G$
  2. **Associativity**: $\forall a, b, c \in G : (a * b) * c = a * (b * c)$
  3. **Identity**: $\exists e \in G$ such that $\forall a \in G : a * e = e * a = a$
  4. **Inverse**: $\forall a \in G, \exists a^{-1} \in G : a * a^{-1} = a^{-1} * a = e$

Abelian Group: Group that is also commutative

- Named after Niels Henrik Abel
- Example: $(\mathbb{Z}, +)$, $(\mathbb{Q} \setminus \{0\}, \times)$

Operation Tables (Cayley Tables):

For finite sets, binary operations can be represented as tables:

```
*  | e  a  b
---|--------
e  | e  a  b
a  | a  b  e
b  | b  e  a
```

This represents the cyclic group $C_3 = \{e, a, b\}$ where $a^2 = b$, $a^3 = e$

Problem-Solving Strategy:

1. **Check closure**: Verify all combinations stay in set
2. **Test associativity**: Check $(a * b) * c = a * (b * c)$ for several triples
3. **Find identity**: Look for element that leaves others unchanged
4. **Find inverses**: For each element, find its inverse (if exists)
5. **Verify axioms systematically**

Common Examples:

- $(\mathbb{Z}, +)$: Abelian group with identity 0, inverse of a is $-a$
- $(\mathbb{Q} \setminus \{0\}, \times)$: Abelian group with identity 1, inverse of a is $1/a$
- $(\mathbb{Z}_n, +_n)$: Integers modulo n under addition modulo n
- Matrix multiplication: Associative but not commutative
- Function composition: Associative but not commutative

GATE Tips:

- Always check closure first - if operation isn't closed, it's not a valid binary operation on that set
- Identity element is unique in any algebraic structure
- In finite groups, every element appears exactly once in each row/column of Cayley table
- Order of group = number of elements
- Lagrange's theorem: Order of subgroup divides order of group

1.2 Countable Uncountable Set (2) {#12-countable-uncountable-set-2}

Key Concepts: Cardinality measures the "size" of sets, even infinite ones. Fundamental distinction between countable and uncountable infinities with profound implications for computability theory.

Cardinality and Equipotence:

Definition: Two sets A and B have the same cardinality (written $|A| = |B|$) if there exists a bijection $f : A \to B$

Finite Cardinality: $|A| = n$ if A can be put in bijection with $\{1, 2, \ldots, n\}$

Infinite Cardinalities:

Countably Infinite: $|A| = |\mathbb{N}| = \aleph_0$ (aleph-null)

- Can be put in bijection with natural numbers
- Elements can be "listed" or "enumerated": $a_1, a_2, a_3, \ldots$

Uncountably Infinite: $|A| > \aleph_0$

- Cannot be put in bijection with natural numbers
- "More" elements than natural numbers

Countable Sets:

Theorem: A set is countable if and only if it is finite or countably infinite

Examples of Countable Sets:

1. Integers $\mathbb{Z}$:
Bijection with $\mathbb{N}$: $f(n) = \begin{cases} n/2 & \text{if } n \text{ even} \\ -(n-1)/2 & \text{if } n \text{ odd} \end{cases}$

Mapping: $0 \mapsto 0, 1 \mapsto -1, 2 \mapsto 1, 3 \mapsto -2, 4 \mapsto 2, \ldots$

2. Rational Numbers $\mathbb{Q}$ (Cantor's Diagonal Argument for Countability):

Arrange positive rationals in array by denominator:

```
1/1  2/1  3/1  4/1  ...
1/2  2/2  3/2  4/2  ...
1/3  2/3  3/3  4/3  ...
1/4  2/4  3/4  4/4  ...
...  ...  ...  ...  ...
```

Traverse diagonally: $1/1, 1/2, 2/1, 3/1, 2/2, 1/3, 1/4, 2/3, 3/2, 4/1, \ldots$

Skip duplicates (like $2/2 = 1/1$) to get enumeration of positive rationals.
Include negatives and zero using similar technique as for integers.

3. Finite Cartesian Products: $\mathbb{N} \times \mathbb{N}$ is countable

Cantor pairing function: $\pi(x, y) = \frac{(x+y)(x+y+1)}{2} + y$

This gives bijection $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

4. Finite Unions: If $A_1, A_2, \ldots, A_n$ are countable, then $\bigcup_{i=1}^{n} A_i$ is countable

5. Countable Unions: If $A_1, A_2, A_3, \ldots$ are countable sets, then $\bigcup_{i=1}^{\infty} A_i$ is countable

Proof: Use "dovetailing" - enumerate elements systematically:

- Round 1: First element from A_1
- Round 2: First from A_2, second from A_1
- Round 3: First from A_3, second from A_2, third from A_1
- Continue diagonally...

Uncountable Sets:

Cantor's Theorem: For any set A, $|A| < |\mathcal{P}(A)|$ (power set has strictly larger cardinality)

Proof: Suppose $f : A \to \mathcal{P}(A)$ is surjective. Define $B = \{x \in A : x \notin f(x)\}$.
Since f is surjective, $\exists a \in A$ such that $f(a) = B$.

- If $a \in B$, then by definition of B, $a \notin f(a) = B$ (contradiction)
- If $a \notin B$, then $a \in f(a) = B$ (contradiction)
  Therefore, no such surjection exists, so $|A| < |\mathcal{P}(A)|$.

Corollary: $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| = |\{0,1\}^{\mathbb{N}}| = 2^{\aleph_0}$

Real Numbers $\mathbb{R}$ are Uncountable (Cantor's Diagonal Argument):

Theorem: $|\mathbb{R}| = 2^{\aleph_0} > \aleph_0$

Proof: Sufficient to show $|(0,1)| > \aleph_0$ (since $|(0,1)| = |\mathbb{R}|$)

Suppose $(0,1)$ is countable with enumeration $r_1, r_2, r_3, \ldots$
Write each in decimal expansion:

```
r₁ = 0.a₁₁a₁₂a₁₃a₁₄...
r₂ = 0.a₂₁a₂₂a₂₃a₂₄...
r₃ = 0.a₃₁a₃₂a₃₃a₃₄...
r₄ = 0.a₄₁a₄₂a₄₃a₄₄...
...
```

Construct $d = 0.d_1 d_2 d_3 d_4 \ldots$ where:
$d_i = \begin{cases} 1 & \text{if } a_{ii} \neq 1 \\ 2 & \text{if } a_{ii} = 1 \end{cases}$

Then d differs from r_i in the i-th decimal place for all i.
So $d \notin \{r_1, r_2, r_3, \ldots\}$, contradicting completeness of enumeration.
Therefore $(0,1)$ is uncountable.

Other Uncountable Sets:

- $\mathcal{P}(\mathbb{N})$ (power set of naturals)
- $\{0,1\}^{\mathbb{N}}$ (infinite binary sequences)
- $\mathbb{R}^n$ for any $n \geq 1$
- Set of all functions $f : \mathbb{N} \to \{0,1\}$
- Cantor set (fractal with cardinality $2^{\aleph_0}$)

Hierarchy of Infinities:

$\aleph_0 < 2^{\aleph_0} < 2^{2^{\aleph_0}} < 2^{2^{2^{\aleph_0}}} < \cdots$

Continuum Hypothesis: $2^{\aleph_0} = \aleph_1$ (next cardinal after $\aleph_0$)

- Independent of ZFC set theory (Gödel, Cohen)
- Cannot be proved or disproved from standard axioms

Problem-Solving Strategy:

To show set is countable:

1. **Explicit bijection**: Construct $f : A \to \mathbb{N}$
2. **Enumeration**: List elements systematically
3. **Use closure properties**: Finite unions, countable unions of countable sets

To show set is uncountable:

1. **Diagonalization**: Assume countable, derive contradiction
2. **Cardinality comparison**: Show $|A| \geq |\mathbb{R}|$ or $|A| \geq 2^{\aleph_0}$
3. **Use Cantor's theorem**: Show $|A| \geq |\mathcal{P}(\mathbb{N})|$

Applications in Computer Science:

- **Computability**: Countably many programs, uncountably many functions → Most functions uncomputable
- **Formal languages**: Some language classes countable, others uncountable
- **Real computation**: Dealing with uncountable domains in continuous mathematics

GATE Tips:

- $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ are countable; $\mathbb{R}, \mathcal{P}(\mathbb{N})$ are uncountable
- Finite unions of countable sets are countable
- Countable unions of countable sets are countable
- Cartesian product of finitely many countable sets is countable
- Any subset of countable set is countable
- Diagonalization is key technique for proving uncountability
- Bijection with $\mathbb{N}$ proves countability

4.3 Functions (29) {#43-functions-29}

Key Concepts: Functions are fundamental mappings between sets that preserve structure and enable mathematical analysis. Understanding their properties is crucial for discrete mathematics and computer science.

Definition: A function $f : A \to B$ is a relation that assigns to each element $a \in A$ exactly one element $b \in B$

- **Domain**: Set A (input set)
- **Codomain**: Set B (target set)
- **Range/Image**: $\text{Im}(f) = \{f(a) : a \in A\} \subseteq B$ (actual outputs)

Function Properties:

1. Injective (One-to-One):

Definition: $f : A \to B$ is injective if $\forall a_1, a_2 \in A : f(a_1) = f(a_2) \Rightarrow a_1 = a_2$

Equivalent formulations:

- Different inputs produce different outputs
- $\forall a_1, a_2 \in A : a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$ (contrapositive)
- Every horizontal line intersects graph at most once

Testing injectivity:

- **Algebraic**: Assume $f(a_1) = f(a_2)$, prove $a_1 = a_2$
- **Graphical**: Horizontal line test
- **Counting**: If $|A| > |B|$, then f cannot be injective (Pigeonhole Principle)

Examples:

- $f : \mathbb{R} \to \mathbb{R}, f(x) = 2x + 3$ is injective
- $f : \mathbb{R} \to \mathbb{R}, f(x) = x^2$ is not injective ($f(-1) = f(1) = 1$)
- $f : [0, \infty) \to \mathbb{R}, f(x) = x^2$ is injective (restricted domain)

2. Surjective (Onto):

Definition: $f : A \to B$ is surjective if $\forall b \in B, \exists a \in A : f(a) = b$

Equivalent formulations:

- Range equals codomain: $\text{Im}(f) = B$
- Every element in codomain has at least one preimage
- Every horizontal line intersects graph at least once

Testing surjectivity:

- **Direct**: For arbitrary $b \in B$, find $a \in A$ such that $f(a) = b$
- **Counting**: If $|A| < |B|$ and both finite, then f cannot be surjective

Examples:

- $f : \mathbb{R} \to \mathbb{R}, f(x) = x^3$ is surjective
- $f : \mathbb{R} \to \mathbb{R}, f(x) = x^2$ is not surjective (negative numbers have no preimage)
- $f : \mathbb{R} \to [0, \infty), f(x) = x^2$ is surjective (restricted codomain)

3. Bijective (One-to-One Correspondence):

Definition: $f : A \to B$ is bijective if it is both injective and surjective

Properties:

- Establishes perfect pairing between sets
- $|A| = |B|$ (same cardinality)
- Every element in B has exactly one preimage in A
- Graph passes horizontal line test exactly once

Fundamental Theorem: $f : A \to B$ is bijective if and only if f has an inverse function $f^{-1} : B \to A$

Inverse Functions:

Definition: If $f : A \to B$ is bijective, then $f^{-1} : B \to A$ is the inverse function where: $f^{-1}(b) = a \Leftrightarrow f(a) = b$

Properties of Inverses:

- $(f^{-1})^{-1} = f$
- $f \circ f^{-1} = \mathrm{id}_B$ (identity on B)
- $f^{-1} \circ f = \mathrm{id}_A$ (identity on A)
- $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$ (reverse order)

Finding Inverse:

1. Write $y = f(x)$
2. Solve for x in terms of y: $x = g(y)$
3. Then $f^{-1}(y) = g(y)$
4. Verify: $f(f^{-1}(y)) = y$ and $f^{-1}(f(x)) = x$

Function Composition:

Definition: $(g \circ f)(x) = g(f(x))$ where $f : A \to B$ and $g : B \to C$

Properties:

- **Associative**: $(h \circ g) \circ f = h \circ (g \circ f)$
- **Not commutative**: Generally $g \circ f \neq f \circ g$
- **Identity**: $f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f$

Composition and Function Properties:

- If f and g are injective, then $g \circ f$ is injective
- If f and g are surjective, then $g \circ f$ is surjective
- If f and g are bijective, then $g \circ f$ is bijective
- If $g \circ f$ is injective, then f is injective
- If $g \circ f$ is surjective, then g is surjective

Special Types of Functions:

1. Identity Function: $\mathrm{id}_A : A \to A$ where $\mathrm{id}_A(a) = a$

- Always bijective
- Neutral element for composition

2. Constant Function: $f : A \to B$ where $f(a) = c$ for all $a \in A$ and some fixed $c \in B$

- Injective only if $|A| \leq 1$
- Surjective only if $|B| = 1$

3. Inclusion Function: $\iota : A \to B$ where $A \subseteq B$ and $\iota(a) = a$

- Always injective
- Surjective iff $A = B$

4. Characteristic Function: $\chi_S : A \to \{0, 1\}$ where $\chi_S(a) = 1$ if $a \in S$, 0 otherwise

- Establishes bijection between subsets of A and functions $A \to \{0, 1\}$

Counting Functions:

For finite sets A with $|A| = m$ and B with $|B| = n$:

Total Functions: n^m (each element in A has n choices)

Injective Functions: $\frac{n!}{(n-m)!} = n(n-1)(n-2)\cdots(n-m+1)$

- Requires $m \leq n$
- First element: n choices, second: $n-1$ choices, etc.

Surjective Functions: $n! \cdot S(m, n)$ where $S(m, n)$ is Stirling number of second kind

- Requires $m \geq n$
- $S(m, n) = \frac{1}{n!} \sum_{k=0}^{n} (-1)^{n-k} \binom{n}{k} k^m$

Bijective Functions: $n!$ (permutations)

- Requires $m = n$

Function Spaces:

Notation: B^A denotes set of all functions from A to B

Cardinality: $|B^A| = |B|^{|A|}$

Examples:

- $\{0, 1\}^{\mathbb{N}}$ represents all infinite binary sequences

- $\mathbb{R}^{\mathbb{R}}$ represents all real-valued functions on reals

Applications in Computer Science:

1. Data Structures: Arrays as functions from indices to values
2. Databases: Relations as functions (in functional dependencies)
3. Cryptography: Hash functions (should behave like random functions)
4. Algorithms: Bijections for counting, injections for avoiding collisions
5. Programming: Function types, higher-order functions

Problem-Solving Strategy:

To prove injectivity:

1. Assume $f(a_1) = f(a_2)$
2. Manipulate algebraically to show $a_1 = a_2$
3. Or use contrapositive: assume $a_1 \neq a_2$, show $f(a_1) \neq f(a_2)$

To prove surjectivity:

1. Take arbitrary $b \in B$
2. Construct or find $a \in A$ such that $f(a) = b$
3. Verify the construction works

To find inverse:

1. Verify function is bijective
2. Solve $y = f(x)$ for x
3. Write $f^{-1}(y) = $ solution
4. Check: $f(f^{-1}(y)) = y$ and $f^{-1}(f(x)) = x$

GATE Tips:

- Pigeonhole principle: If $|A| > |B|$, no injection $A \to B$ exists
- If $|A| < |B|$, no surjection $A \to B$ exists (finite sets)
- Composition reverses order for inverses: $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$
- Function has inverse iff bijective
- For finite sets: bijection iff $|A| = |B|$ and function is injective (or surjective)
- Identity function is always bijective
- Constant function rarely bijective (only when domain and codomain have size 1)

Examples with Analysis:

Example 1: $f : \mathbb{R} \to \mathbb{R}, f(x) = x^2$

- **Injective?** No: $f(-2) = f(2) = 4$
- **Surjective?** No: -1 has no preimage

- **Bijective?** No

Example 2: $f : [0, \infty) \to [0, \infty), f(x) = x^2$

- **Injective?** Yes: If $x_1^2 = x_2^2$ and $x_1, x_2 \geq 0$, then $x_1 = x_2$
- **Surjective?** Yes: For any $y \geq 0$, take $x = \sqrt{y}$
- **Bijective?** Yes
- **Inverse**: $f^{-1}(y) = \sqrt{y}$

Example 3: $f : \{1, 2, 3\} \to \{a, b\}, f(1) = a, f(2) = a, f(3) = b$

- **Injective?** No: $f(1) = f(2)$
- **Surjective?** Yes: Both a and b have preimages
- **Bijective?** No

Example 4: Counting injective functions from 4-element set to 6-element set

- Answer: $6 \cdot 5 \cdot 4 \cdot 3 = 360$
- Or: $\frac{6!}{(6-4)!} = \frac{6!}{2!} = 360$

4.4 Group Theory (33) {#44-group-theory-33}

Key Concepts: Group theory studies algebraic structures with a single associative operation, identity, and inverses. Fundamental to abstract algebra, cryptography, and symmetry analysis.

Group Definition:

A **group** $(G, *)$ is a set G with binary operation $*$ satisfying:

1. **Closure**: $\forall a, b \in G : a * b \in G$
2. **Associativity**: $\forall a, b, c \in G : (a * b) * c = a * (b * c)$
3. **Identity**: $\exists e \in G$ such that $\forall a \in G : a * e = e * a = a$
4. **Inverse**: $\forall a \in G, \exists a^{-1} \in G : a * a^{-1} = a^{-1} * a = e$

Basic Properties:

Uniqueness of Identity: If e_1 and e_2 are both identities, then $e_1 = e_1 * e_2 = e_2$

Uniqueness of Inverse: If b and c are both inverses of a, then:
$b = b * e = b * (a * c) = (b * a) * c = e * c = c$

Cancellation Laws:

- **Left cancellation**: $a * b = a * c \Rightarrow b = c$
- **Right cancellation**: $b * a = c * a \Rightarrow b = c$

Proof of left cancellation: $a * b = a * c \Rightarrow a^{-1} * (a * b) = a^{-1} * (a * c) \Rightarrow b = c$

Order of Group: $|G|$ = number of elements in G

- **Finite group**: $|G| < \infty$
- **Infinite group**: $|G| = \infty$

Examples of Groups:

1. Additive Groups:

- $(\mathbb{Z}, +)$: Integers under addition, identity 0, inverse of a is $-a$
- $(\mathbb{Q}, +)$, $(\mathbb{R}, +)$, $(\mathbb{C}, +)$: Rational, real, complex numbers under addition
- $(\mathbb{Z}_n, +_n)$: Integers modulo n, order n

2. Multiplicative Groups:

- $(\mathbb{Q}^*, \times)$: Non-zero rationals under multiplication, identity 1
- $(\mathbb{R}^*, \times)$, $(\mathbb{C}^*, \times)$: Non-zero reals, complex numbers
- $(\mathbb{Z}_p^*, \times_p)$: Non-zero integers modulo prime p, order $p-1$

3. Matrix Groups:

- $GL_n(\mathbb{R})$: Invertible $n \times n$ real matrices under multiplication
- $SL_n(\mathbb{R})$: Matrices with determinant 1
- $O_n(\mathbb{R})$: Orthogonal matrices ($A^T A = I$)

4. Symmetric Groups:

- S_n: All permutations of n elements, order $n!$
- A_n: Even permutations (alternating group), order $n!/2$

5. Cyclic Groups:

- $C_n = \langle a \rangle = \{e, a, a^2, \ldots, a^{n-1}\}$ where $a^n = e$
- Every finite cyclic group isomorphic to $\mathbb{Z}_n$
- Every infinite cyclic group isomorphic to $\mathbb{Z}$

Subgroups:

Definition: $H \subseteq G$ is a **subgroup** (written $H \leq G$) if:

1. H is non-empty
2. H is closed under the group operation
3. H contains identity of G
4. H is closed under taking inverses

Subgroup Test: $H \leq G$ iff $H \neq \emptyset$ and $\forall a, b \in H : ab^{-1} \in H$

Examples of Subgroups:

- $\{e\}$ and G are **trivial subgroups**
- $n\mathbb{Z} = \{nk : k \in \mathbb{Z}\} \leq \mathbb{Z}$
- $A_n \leq S_n$ (even permutations)
- $SL_n(\mathbb{R}) \leq GL_n(\mathbb{R})$

Generated Subgroups:

Definition: For $a \in G$, the **cyclic subgroup generated by** a is:
$\langle a \rangle = \{a^n : n \in \mathbb{Z}\}$

Order of Element: $|a| = |\langle a \rangle|$ = smallest positive integer n such that $a^n = e$

- If no such n exists, then $|a| = \infty$

Properties:

- $|a^k| = \frac{|a|}{\gcd(|a|,k)}$
- $a^m = e \Leftrightarrow |a|$ divides m
- In finite group, $|a|$ divides $|G|$ (by Lagrange's theorem)

Lagrange's Theorem:

Theorem: If G is finite group and $H \leq G$, then $|H|$ divides $|G|$

Corollary: $|G| = |H| \cdot [G : H]$ where $[G : H]$ is the **index** of H in G

Proof Outline:

1. Define **left cosets**: $aH = \{ah : h \in H\}$ for $a \in G$
2. Show cosets partition G: $G = \bigcup_{a \in G} aH$ (disjoint union)
3. All cosets have same size: $|aH| = |H|$
4. Number of cosets = $[G : H] = |G|/|H|$

Corollaries of Lagrange:

- **Order of element divides order of group**: $|a|$ divides $|G|$
- **Fermat's Little Theorem**: If p prime and $\gcd(a, p) = 1$, then $a^{p-1} \equiv 1 \pmod{p}$
- **Euler's Theorem**: If $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$

Cyclic Groups:

Definition: Group G is **cyclic** if $\exists a \in G$ such that $G = \langle a \rangle$

- Element a is called a **generator**

Properties of Cyclic Groups:

- Every cyclic group is abelian

- Every subgroup of cyclic group is cyclic
- $\mathbb{Z}_n$ is cyclic with generators $\{a : \gcd(a, n) = 1\}$
- Number of generators of $\mathbb{Z}_n$ is $\phi(n)$ (Euler's totient function)

Classification of Cyclic Groups:

- **Finite cyclic**: $C_n \cong \mathbb{Z}_n$
- **Infinite cyclic**: $C_\infty \cong \mathbb{Z}$

Subgroups of Cyclic Groups:

Theorem: Every subgroup of $\mathbb{Z}$ has form $n\mathbb{Z}$ for some $n \geq 0$

Theorem: Subgroups of $\mathbb{Z}_n$ are in bijection with divisors of n:

- For each divisor d of n, there is unique subgroup of order d
- This subgroup is $\langle n/d \rangle = \{0, n/d, 2n/d, \ldots, (d-1)n/d\}$

Example: Subgroups of $\mathbb{Z}_{12}$:

- Divisors of 12: $1, 2, 3, 4, 6, 12$
- Subgroups:
  - Order 1: $\{0\}$
  - Order 2: $\{0, 6\} = \langle 6 \rangle$
  - Order 3: $\{0, 4, 8\} = \langle 4 \rangle$
  - Order 4: $\{0, 3, 6, 9\} = \langle 3 \rangle$
  - Order 6: $\{0, 2, 4, 6, 8, 10\} = \langle 2 \rangle$
  - Order 12: $\mathbb{Z}_{12} = \langle 1 \rangle$

Group Homomorphisms:

Definition: Function $\phi : G \to H$ is **homomorphism** if:
$\forall a, b \in G : \phi(ab) = \phi(a)\phi(b)$

Properties:

- $\phi(e_G) = e_H$ (identity maps to identity)
- $\phi(a^{-1}) = \phi(a)^{-1}$ (inverse maps to inverse)
- $\phi(a^n) = \phi(a)^n$ for all integers n

Types of Homomorphisms:

- **Monomorphism**: Injective homomorphism
- **Epimorphism**: Surjective homomorphism
- **Isomorphism**: Bijective homomorphism (groups are "essentially same")
- **Endomorphism**: Homomorphism from group to itself

- **Automorphism**: Isomorphism from group to itself

Kernel and Image:

- **Kernel**: $\ker(\phi) = \{g \in G : \phi(g) = e_H\}$
- **Image**: $\text{Im}(\phi) = \{\phi(g) : g \in G\}$

Properties:

- $\ker(\phi) \leq G$ (always a subgroup)
- $\text{Im}(\phi) \leq H$
- ϕ is injective iff $\ker(\phi) = \{e_G\}$
- $|G| = |\ker(\phi)| \cdot |\text{Im}(\phi)|$ (for finite groups)

Isomorphism Theorems:

First Isomorphism Theorem: If $\phi : G \to H$ is homomorphism, then:
$G/\ker(\phi) \cong \text{Im}(\phi)$

Normal Subgroups and Quotient Groups:

Definition: $N \leq G$ is **normal** (written $N \triangleleft G$) if:
$\forall g \in G : gNg^{-1} = N$

Equivalently: $\forall g \in G, n \in N : gng^{-1} \in N$

Properties:

- In abelian groups, every subgroup is normal
- Kernel of any homomorphism is normal
- Index-2 subgroups are always normal

Quotient Group: If $N \triangleleft G$, then $G/N = \{gN : g \in G\}$ forms group under:
$(aN)(bN) = (ab)N$

Applications in Computer Science:

1. Cryptography:

- RSA: Uses multiplicative group $\mathbb{Z}_n^*$
- Elliptic curve cryptography: Uses elliptic curve groups
- Discrete logarithm problem in cyclic groups

2. Error-Correcting Codes:

- Linear codes form groups under addition
- Syndrome decoding uses cosets

3. Computer Graphics:

- Rotation groups for 3D transformations
- Symmetry groups for geometric objects

4. Algorithms:

- Group-theoretic algorithms for graph isomorphism
- Fast Fourier Transform uses cyclic groups

Problem-Solving Strategy:

To verify group:

1. Check closure (often use operation table for finite groups)
2. Verify associativity (use known associative operations when possible)
3. Find identity element
4. Find inverse for each element

To find subgroups:

1. Use subgroup test: non-empty, closed under ab^{-1}
2. For cyclic groups: find divisors of order
3. Look for kernels of homomorphisms

To apply Lagrange's theorem:

1. Find order of group and subgroup
2. Check divisibility
3. Use to find possible orders of elements/subgroups

GATE Tips:

- Order of element divides order of group (Lagrange)
- In $\mathbb{Z}_n$: element a has order $n/\gcd(a, n)$
- Cyclic group of order n has $\phi(n)$ generators
- Every group of prime order is cyclic
- Subgroups of $\mathbb{Z}_n$ correspond to divisors of n
- Abelian groups: all subgroups normal, quotients well-defined
- Homomorphism preserves structure: $\phi(ab) = \phi(a)\phi(b)$
- Kernel measures "failure to be injective"

Common Examples for GATE:

Example 1: $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$ under addition mod 6

- Subgroups: $\{0\}$, $\{0, 2, 4\}$, $\{0, 3\}$, $\mathbb{Z}_6$
- Orders: 1, 3, 2, 6 (all divide 6)

- Generators: 1 and 5 (since $\gcd(1,6) = \gcd(5,6) = 1$)

Example 2: S_3 (symmetric group on 3 elements)

- Elements: $e, (12), (13), (23), (123), (132)$
- Order: 6
- Subgroups: $\{e\}$, $\{e, (12)\}$, $\{e, (13)\}$, $\{e, (23)\}$, $\{e, (123), (132)\}$, S_3
- $A_3 = \{e, (123), (132)\}$ is normal subgroup

Example 3: Multiplicative group $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$

- Order: 6
- Cyclic (since 7 is prime)
- Generators: elements with order 6
- $\phi(6) = 2$, so 2 generators: check which elements have order 6

4.5 Identity Function (1) {#45-identity-function-1}

Key Concepts: $textid(x) = x$, bijective. **Formulas/Theorems**: Always automorphism. **Problem-Solving Tips**: Trivial in compositions.

4.6 Lattice (10) {#46-lattice-10}

Key Concepts: A lattice is an algebraic structure with two binary operations (meet and join) satisfying specific properties. Fundamental in order theory, logic, and computer science.

Partial Order (Poset):

Definition: A relation $\leq$ on set S is a **partial order** if:

1. **Reflexive**: $a \leq a$ for all $a \in S$
2. **Antisymmetric**: If $a \leq b$ and $b \leq a$, then $a = b$
3. **Transitive**: If $a \leq b$ and $b \leq c$, then $a \leq c$

Lattice Definition:

Definition: A **lattice** $(L, \leq)$ is a poset where every pair of elements has:

- **Greatest lower bound (GLB)**: $a \wedge b$ (meet)
- **Least upper bound (LUB)**: $a \vee b$ (join)

Algebraic Definition: $(L, \wedge, \vee)$ where $\wedge$ (meet) and $\vee$ (join) satisfy:

Idempotent Laws:

- $a \wedge a = a$
- $a \vee a = a$

Commutative Laws:

- $a \wedge b = b \wedge a$
- $a \vee b = b \vee a$

Associative Laws:

- $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
- $(a \vee b) \vee c = a \vee (b \vee c)$

Absorption Laws:

- $a \wedge (a \vee b) = a$
- $a \vee (a \wedge b) = a$

Connection to Order: $a \leq b \Leftrightarrow a \wedge b = a \Leftrightarrow a \vee b = b$

Bounded Lattice:

Definition: Lattice with:

- **Top element** $\top$ (greatest): $a \leq \top$ for all a
- **Bottom element** $\bot$ (least): $\bot \leq a$ for all a

Properties:

- $a \vee \top = \top$
- $a \wedge \bot = \bot$
- $a \vee \bot = a$
- $a \wedge \top = a$

Distributive Lattice:

Definition: Lattice where meet distributes over join (and vice versa):

- $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

Theorem: If one distributive law holds, the other holds automatically.

Complemented Lattice:

Definition: Bounded lattice where every element a has **complement** a' such that:

- $a \wedge a' = \bot$
- $a \vee a' = \top$

Note: Complements need not be unique in general lattices

Boolean Algebra:

Definition: A **Boolean algebra** is a complemented distributive lattice

Properties:

- Complements are unique
- **De Morgan's Laws**:
  - $(a \wedge b)' = a' \vee b'$
  - $(a \vee b)' = a' \wedge b'$
- **Double complement**: $(a')' = a$
- **Complement laws**:
  - $\top' = \bot$
  - $\bot' = \top$

Examples of Lattices:

1. Power Set Lattice $\mathcal{P}(S)$:

- Order: $\subseteq$ (subset relation)
- Meet: $A \wedge B = A \cap B$ (intersection)
- Join: $A \vee B = A \cup B$ (union)
- Top: S
- Bottom: $\emptyset$
- Distributive and complemented (Boolean algebra)
- Complement: $A' = S \setminus A$

2. Divisibility Lattice on positive integers:

- Order: $a \leq b$ if $a \mid b$
- Meet: $a \wedge b = \gcd(a, b)$
- Join: $a \vee b = \operatorname{lcm}(a, b)$
- Bottom: 1 (divides everything)
- No top element (unbounded)
- Distributive but not complemented

3. Subset of Divisors D_n of integer n:

- Order: divisibility
- Meet: GCD
- Join: LCM
- Top: n
- Bottom: 1
- Distributive

4. Linear Orders (Total Orders):

- Any two elements comparable
- Examples: $\mathbb{R}, \mathbb{Z}, \mathbb{N}$ with $\leq$
- Meet: $\min(a, b)$
- Join: $\max(a, b)$
- Distributive

5. Function Lattice:

- Functions $f : X \to L$ where L is lattice
- Pointwise operations: $(f \wedge g)(x) = f(x) \wedge g(x)$
- Inherits lattice properties from L

Hasse Diagrams:

Definition: Graphical representation of finite poset

- Vertices: elements
- Edges: cover relations (a covers b if $a > b$ and no c with $a > c > b$)
- Draw smaller elements below larger ones
- Omit reflexive and transitive edges

Example: Divisors of 12

Reading Hasse Diagrams:

- Meet: Greatest common lower bound (go down)
- Join: Least common upper bound (go up)
- $a \leq b$: Path from a upward to b

Complete Lattice:

Definition: Lattice where every subset has GLB and LUB

- Implies bounded (empty set has GLB = $\top$, LUB = $\bot$)
- Finite lattices are always complete

Fixed Point Theorems:

Knaster-Tarski Theorem: If $f : L \to L$ is monotone function on complete lattice, then set of fixed points forms complete lattice

Application: Least fixed point semantics in programming languages

Modular Lattice:

Definition: Lattice satisfying **modular law**:
If $a \leq c$, then $a \vee (b \wedge c) = (a \vee b) \wedge c$

Properties:

- Every distributive lattice is modular
- Not all modular lattices are distributive
- Example: Subspace lattice of vector space is modular but not always distributive

Lattice Homomorphisms:

Definition: Function $f : L_1 \to L_2$ between lattices is **homomorphism** if:

- $f(a \wedge b) = f(a) \wedge f(b)$
- $f(a \vee b) = f(a) \vee f(b)$

Isomorphism: Bijective homomorphism with inverse also homomorphism

Applications in Computer Science:

1. Data Flow Analysis:

- Lattice of program properties
- Meet: Intersection of properties
- Fixed point iteration for analysis

2. Type Systems:

- Subtyping forms lattice
- Meet: Greatest common subtype
- Join: Least common supertype

3. Formal Concept Analysis:

- Concept lattice from data
- Applications in data mining

4. Domain Theory:

- Scott domains for denotational semantics
- Continuous lattices for recursion

5. Concurrency Theory:

- Event structures
- Partial order models

Problem-Solving Strategy:

To verify lattice:

1. Check partial order properties
2. For each pair, find meet and join
3. Verify uniqueness of GLB and LUB

To check distributivity:

1. Test distributive law on several triples
2. Look for pentagon N_5 or diamond M_3 sublattices (non-distributive)

To find complements:

1. For each element a, find b where $a \wedge b = \bot$ and $a \vee b = \top$
2. In Boolean algebra, complement is unique

GATE Tips:

- Power set with $\subseteq$ is always Boolean algebra
- Divisibility lattice: meet = GCD, join = LCM
- Distributive lattice: meet distributes over join
- Boolean algebra = complemented distributive lattice
- Hasse diagram: draw smaller elements below
- Complete lattice: every subset has GLB and LUB
- Linear order: any two elements comparable
- Modular lattice: weaker than distributive

Common GATE Examples:

Example 1: Lattice of divisors of 30

- Elements: {1, 2, 3, 5, 6, 10, 15, 30}
- $2 \wedge 3 = \gcd(2, 3) = 1$
- $2 \vee 3 = \text{lcm}(2, 3) = 6$
- Distributive: Yes
- Complemented: No (e.g., 2 has no complement)

Example 2: Power set of {a,b}

- Elements: $\emptyset, \{a\}, \{b\}, \{a, b\}$

- $\{a\} \wedge \{b\} = \emptyset$
- $\{a\} \vee \{b\} = \{a, b\}$
- Boolean algebra: Yes
- Complement of $\{a\}$ is $\{b\}$

Example 3: Check distributivity
For lattice with elements {0, a, b, c, 1} where a, b, c incomparable:

- If $a \wedge (b \vee c) \neq (a \wedge b) \vee (a \wedge c)$, not distributive
- Pentagon N_5 or diamond M_3 indicate non-distributive

4.7 Mathematical Induction (2) {#47-mathematical-induction-2}

Key Concepts: Mathematical induction is a fundamental proof technique for statements involving natural numbers. It establishes truth for infinitely many cases using finite verification.

Principle of Mathematical Induction:

To prove statement $P(n)$ is true for all $n \geq n_0$:

1. **Base Case**: Prove $P(n_0)$ is true
2. **Inductive Step**: Prove $P(k) \Rightarrow P(k+1)$ for arbitrary $k \geq n_0$
3. **Conclusion**: $P(n)$ is true for all $n \geq n_0$

Logical Foundation:

Based on **Well-Ordering Principle**: Every non-empty subset of natural numbers has a least element.

Proof of Induction Principle:
Suppose $P(n_0)$ true and $P(k) \Rightarrow P(k+1)$ for all $k \geq n_0$.
Assume for contradiction that $P(n)$ is false for some $n \geq n_0$.
Let $S = \{n \geq n_0 : P(n) \text{ is false}\}$.
By assumption, $S \neq \emptyset$, so by well-ordering, S has least element m.
Since $P(n_0)$ is true, $m > n_0$, so $m - 1 \geq n_0$.
Since m is least element of S, $P(m-1)$ is true.
By inductive step, $P(m-1) \Rightarrow P(m)$, so $P(m)$ is true.
Contradiction! Therefore $S = \emptyset$, so $P(n)$ true for all $n \geq n_0$.

Types of Induction:

1. Weak (Simple) Induction:

- Assume $P(k)$ true, prove $P(k+1)$ true
- **Inductive Hypothesis**: $P(k)$ is true
- **Template**:

```
Base Case: Show P(n₀) is true
Inductive Step:
  Assume P(k) is true for some k ≥ n₀
  Show P(k+1) is true
Conclusion: P(n) true for all n ≥ n₀
```

2. Strong Induction:

- Assume $P(n_0), P(n_0+1), \ldots, P(k)$ all true, prove $P(k+1)$ true
- **Inductive Hypothesis**: $P(j)$ is true for all $n_0 \leq j \leq k$
- **When to use**: When $P(k+1)$ depends on multiple previous cases
- **Template**:

```
Base Case(s): Show P(n₀), P(n₀+1), ..., P(n₀+r-1) true
Inductive Step:
  Assume P(j) true for all n₀ ≤ j ≤ k (where k ≥ n₀+r-1)
  Show P(k+1) is true
Conclusion: P(n) true for all n ≥ n₀
```

3. Structural Induction:

- For recursively defined structures (trees, expressions, etc.)
- Base case: Prove for atomic elements
- Inductive step: If true for components, prove true for composite

Classic Examples:

Example 1: Sum Formula
Claim: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ for all $n \geq 1$

Proof:

- **Base Case** ($n = 1$): $\sum_{k=1}^{1} k = 1$ and $\frac{1(1+1)}{2} = 1$ ✓
- **Inductive Step**: Assume true for $n = m$: $\sum_{k=1}^{m} k = \frac{m(m+1)}{2}$

  Need to prove for $n = m + 1$:

$$\sum_{k=1}^{m+1} k = \sum_{k=1}^{m} k + (m+1) = \frac{m(m+1)}{2} + (m+1)$$

$$= \frac{m(m+1) + 2(m+1)}{2} = \frac{(m+1)(m+2)}{2}$$

  This matches the formula for $n = m + 1$ ✓
- **Conclusion**: Formula holds for all $n \geq 1$

Example 2: Geometric Series
Claim: $\sum_{k=0}^{n} r^k = \frac{r^{n+1}-1}{r-1}$ for $r \neq 1$, $n \geq 0$

Proof:

- **Base Case** ($n = 0$): $\sum_{k=0}^{0} r^k = r^0 = 1$ and $\frac{r^1-1}{r-1} = \frac{r-1}{r-1} = 1$ ✓
- **Inductive Step**: Assume true for $n = m$: $\sum_{k=0}^{m} r^k = \frac{r^{m+1}-1}{r-1}$

  For $n = m + 1$:

$$\sum_{k=0}^{m+1} r^k = \sum_{k=0}^{m} r^k + r^{m+1} = \frac{r^{m+1} - 1}{r - 1} + r^{m+1}$$

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

  This matches the formula for $n = m + 1$ ✓

Example 3: Inequality
Claim: $2^n > n^2$ for all $n \geq 5$

Proof:

- **Base Case** ($n = 5$): $2^5 = 32 > 25 = 5^2$ ✓
- **Inductive Step**: Assume $2^m > m^2$ for some $m \geq 5$

  Need to show $2^{m+1} > (m+1)^2$:

$$2^{m+1} = 2 \cdot 2^m > 2m^2 \text{ (by inductive hypothesis)}$$

  Need: $2m^2 > (m+1)^2 = m^2 + 2m + 1$

  This is equivalent to: $m^2 > 2m + 1$, or $m^2 - 2m - 1 > 0$

  For $m \geq 5$: $m^2 - 2m - 1 = m(m-2) - 1 \geq 5(3) - 1 = 14 > 0$ ✓

Example 4: Strong Induction - Fibonacci
Claim: Every integer $n \geq 1$ can be written as sum of distinct Fibonacci numbers

Proof (using strong induction):

- **Base Cases**: $1 = F_2$, $2 = F_3$, $3 = F_4$, $4 = F_2 + F_4$ ✓
- **Inductive Step**: Assume claim true for all $1 \leq j \leq k$ where $k \geq 4$

  For $k + 1$: Let F_m be largest Fibonacci number $\leq k + 1$

  Then $k + 1 - F_m < F_{m-1}$ (by Fibonacci property)

  So $k + 1 - F_m \leq k$, and by inductive hypothesis, $k + 1 - F_m$ can be written as sum of distinct Fibonacci numbers, none equal to F_m

  Therefore $k + 1 = F_m + (\text{sum for } k + 1 - F_m)$ ✓

Common Proof Patterns:

1. Summation Formulas:

- Use algebraic manipulation in inductive step
- Factor out common terms
- Often involves completing the square or factoring

2. Inequalities:

- May need to strengthen inductive hypothesis
- Use properties of functions (monotonicity, convexity)
- Sometimes require multiple base cases

3. Divisibility:

- Use modular arithmetic
- Factor expressions to show divisibility
- Example: $n^3 - n$ is divisible by 6 for all $n \geq 1$

4. Recurrence Relations:

- Strong induction often needed
- Base cases match initial conditions
- Example: Proving closed form for Fibonacci numbers

5. Graph Theory:

- Induction on number of vertices or edges
- Example: Tree with n vertices has $n - 1$ edges

Common Mistakes:

1. Incomplete Base Case:

- Must verify all necessary initial values
- For strong induction, may need multiple base cases

2. Assuming What You Want to Prove:

- In inductive step, only assume $P(k)$, not $P(k+1)$
- Don't use the conclusion in the proof

3. Incorrect Inductive Hypothesis:

- Make sure hypothesis is strong enough
- Sometimes need to prove stronger statement

4. Gaps in Logic:

- Every step must be justified
- Don't skip algebraic manipulations

5. Wrong Direction:

- Prove $P(k) \Rightarrow P(k+1)$, not $P(k+1) \Rightarrow P(k)$

Problem-Solving Strategy:

1. Identify the Pattern:

- What exactly are you trying to prove?
- What is the variable being inducted on?
- What is the base case?

2. Choose Induction Type:

- **Weak induction**: If $P(k+1)$ depends only on $P(k)$
- **Strong induction**: If $P(k+1)$ depends on multiple previous cases

3. Prove Base Case(s):

- Verify smallest value(s) explicitly
- Don't assume - calculate directly

4. Set Up Inductive Step:

- State inductive hypothesis clearly
- Identify what needs to be proven

5. Bridge the Gap:

- Connect $P(k)$ to $P(k+1)$ using valid reasoning
- Use algebraic manipulation, known inequalities, etc.

6. Conclude:

- State that by mathematical induction, the result holds

Applications in Computer Science:

1. Algorithm Correctness:

- Prove loop invariants
- Verify recursive algorithms
- Example: Correctness of merge sort

2. Data Structure Properties:

- Height of balanced trees
- Properties of heaps
- Example: AVL tree height is $O(\log n)$

3. Complexity Analysis:

- Solving recurrence relations
- Proving time/space bounds
- Example: $T(n) = 2T(n/2) + n$ gives $T(n) = O(n \log n)$

4. Formal Verification:

- Program correctness proofs
- Protocol verification
- Hardware verification

5. Combinatorial Arguments:

- Counting problems
- Graph coloring
- Network flows

GATE Tips:

- Always verify base case explicitly - don't assume
- In inductive step, clearly state what you're assuming vs. what you're proving
- For inequalities, may need to prove stronger statement than asked
- Strong induction useful for recurrence relations and when multiple previous cases needed
- Common to prove formulas for sums, products, inequalities
- Watch for off-by-one errors in indexing
- Sometimes need to start induction from $n = 0$ vs. $n = 1$
- For divisibility problems, use modular arithmetic
- Graph theory problems often use induction on vertices/edges

Template for GATE Answers:

```
Proof by Mathematical Induction:

Base Case (n = n₀):
[Verify P(n₀) directly]

Inductive Step:
Assume P(k) is true for some k ≥ n₀.
[State what P(k) means]
```

```
We need to prove P(k+1):
[State what P(k+1) means]

[Algebraic manipulation using inductive hypothesis]

Therefore P(k+1) is true.

Conclusion:
By mathematical induction, P(n) is true for all n ≥ n₀.
```

4.8 Number Theory (7) {#48-number-theory-7}

Key Concepts: Number theory studies properties of integers, focusing on divisibility, primes, and modular arithmetic. Foundation for cryptography, computer algebra, and discrete mathematics.

Divisibility:

Definition: Integer a **divides** integer b (written $a \mid b$) if $\exists k \in \mathbb{Z}$ such that $b = ak$

Properties of Divisibility:

1. **Reflexive**: $a \mid a$ for all $a \neq 0$
2. **Transitive**: If $a \mid b$ and $b \mid c$, then $a \mid c$
3. **Linear combination**: If $a \mid b$ and $a \mid c$, then $a \mid (bx + cy)$ for any integers x, y
4. **Antisymmetric**: If $a \mid b$ and $b \mid a$, then $|a| = |b|$

Division Algorithm:

Theorem: For integers a and b with $b > 0$, there exist unique integers q (quotient) and r (remainder) such that:

$$a = bq + r \text{ where } 0 \leq r < b$$

Notation: $q = \lfloor a/b \rfloor$ and $r = a \bmod b$

Proof Existence: Consider set $S = \{a - bk : k \in \mathbb{Z}, a - bk \geq 0\}$
By well-ordering principle, S has minimum element $r = a - bq$
If $r \geq b$, then $a - b(q + 1) = r - b \geq 0$, contradicting minimality
Therefore $0 \leq r < b$

Proof Uniqueness: If $a = bq_1 + r_1 = bq_2 + r_2$ with $0 \leq r_1, r_2 < b$
Then $b(q_1 - q_2) = r_2 - r_1$
Since $|r_2 - r_1| < b$ and $b \mid (r_2 - r_1)$, we have $r_2 - r_1 = 0$
Therefore $r_1 = r_2$ and $q_1 = q_2$

Greatest Common Divisor (GCD):

Definition: $\gcd(a, b)$ is the largest positive integer that divides both a and b

Properties:

1. $\gcd(a, b) = \gcd(b, a)$ (commutative)
2. $\gcd(a, b) = \gcd(|a|, |b|)$ (absolute values)
3. $\gcd(a, 0) = |a|$ for $a \neq 0$
4. $\gcd(a, b) = \gcd(a - bk, b)$ for any integer k
5. If $d \mid a$ and $d \mid b$, then $d \mid \gcd(a, b)$

Euclidean Algorithm:

Algorithm: To find $\gcd(a, b)$ where $a \geq b > 0$:

```
while b ≠ 0:
    r = a mod b
    a = b
    b = r
return a
```

Correctness: Based on property $\gcd(a, b) = \gcd(b, a \bmod b)$

Proof: Let $d = \gcd(a, b)$ and $r = a \bmod b$, so $a = bq + r$

- If $c \mid b$ and $c \mid r$, then $c \mid (bq + r) = a$, so $c \mid \gcd(a, b)$
- If $c \mid a$ and $c \mid b$, then $c \mid (a - bq) = r$, so $c \mid \gcd(b, r)$
- Therefore $\gcd(a, b) = \gcd(b, r)$

Time Complexity: $O(\log \min(a, b))$

Proof: If $a \geq b$, then $a \bmod b < a/2$
So values decrease by factor of 2 every 2 steps
Number of steps is $O(\log \min(a, b))$

Example: $\gcd(1071, 462)$

```
1071 = 462 × 2 + 147
462 = 147 × 3 + 21
147 = 21 × 7 + 0
```

Therefore $\gcd(1071, 462) = 21$

Extended Euclidean Algorithm:

Bézout's Identity: For integers a, b, there exist integers x, y such that:

$$ax + by = \gcd(a, b)$$

Algorithm: Maintains invariant that at each step:
$a = s_i \cdot a_0 + t_i \cdot b_0$ and $b = s_{i+1} \cdot a_0 + t_{i+1} \cdot b_0$

```
ExtendedGCD(a, b):
    if b = 0:
        return (a, 1, 0)  // gcd, x, y
    else:
        (d, x1, y1) = ExtendedGCD(b, a mod b)
        x = y1
        y = x1 - (a div b) × y1
        return (d, x, y)
```

Example: Find x, y such that $1071x + 462y = \gcd(1071, 462) = 21$

Working backwards from Euclidean algorithm:

```
21 = 462 - 147 × 3
   = 462 - (1071 - 462 × 2) × 3
   = 462 - 1071 × 3 + 462 × 6
   = 462 × 7 - 1071 × 3
```

So $x = -3, y = 7$: $1071(-3) + 462(7) = -3213 + 3234 = 21$ ✓

Applications of Extended GCD:

1. **Modular inverse**: $a^{-1} \bmod m$ exists iff $\gcd(a, m) = 1$
2. **Linear Diophantine equations**: $ax + by = c$ has solution iff $\gcd(a, b) \mid c$
3. **Chinese Remainder Theorem**: Solving systems of congruences

Prime Numbers:

Definition: Integer $p > 1$ is **prime** if its only positive divisors are 1 and p

Composite: Integer $n > 1$ that is not prime

Fundamental Theorem of Arithmetic:

Theorem: Every integer $n > 1$ can be written uniquely (up to order) as product of primes:

$$n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$$

where $p_1 < p_2 < \cdots < p_k$ are primes and $a_i > 0$

Proof Outline:

- **Existence**: By strong induction. If n prime, done. If composite, $n = ab$ where $1 < a, b < n$. By induction, a and b have prime factorizations.
- **Uniqueness**: Suppose $n = p_1^{a_1} \cdots p_r^{a_r} = q_1^{b_1} \cdots q_s^{b_s}$. Since $p_1 \mid n$, we have $p_1 \mid q_1^{b_1} \cdots q_s^{b_s}$. By Euclid's lemma, $p_1 \mid q_j$ for some j. Since q_j prime, $p_1 = q_j$. Continue by induction.

Euclid's Lemma: If prime $p \mid ab$, then $p \mid a$ or $p \mid b$

Corollary: If $p \mid a_1 a_2 \cdots a_n$, then $p \mid a_i$ for some i

Prime Distribution:

Euclid's Theorem: There are infinitely many primes

Proof: Suppose finitely many primes $p_1, p_2, \ldots, p_k$
Consider $N = p_1 p_2 \cdots p_k + 1$
$N > 1$, so has prime divisor p
If $p = p_i$ for some i, then $p \mid N$ and $p \mid (N - 1)$, so $p \mid 1$ (impossible)
Therefore p is new prime not in list

Prime Number Theorem: $\pi(x) \sim \frac{x}{\ln x}$ where $\pi(x)$ counts primes $\leq x$

Sieve of Eratosthenes: Algorithm to find all primes up to n

1. List numbers 2 to n
2. Start with $p = 2$
3. Mark all multiples of p (except p itself)
4. Find next unmarked number, set as new p
5. Repeat until $p^2 > n$
6. Remaining unmarked numbers are prime

Time Complexity: $O(n \log \log n)$

Primality Testing:

Trial Division: Test divisibility by all numbers up to $\sqrt{n}$

- Time: $O(\sqrt{n})$
- Sufficient since if $n = ab$ with $a, b > \sqrt{n}$, then $ab > n$

Fermat's Little Theorem: If p prime and $\gcd(a, p) = 1$, then $a^{p-1} \equiv 1 \pmod{p}$

Fermat Primality Test: If $a^{n-1} \not\equiv 1 \pmod{n}$, then n composite

- **Carmichael numbers**: Composite numbers that pass test for all a coprime to n
- Example: 561 = 3 × 11 × 17

Miller-Rabin Test: Probabilistic primality test with error probability $< 1/4$ per round

Modular Arithmetic:

Congruence: $a \equiv b \pmod{m}$ means $m \mid (a - b)$

Properties:

1. **Equivalence relation**: Reflexive, symmetric, transitive
2. **Addition**: $a \equiv b \pmod{m} \Rightarrow a + c \equiv b + c \pmod{m}$

3. **Multiplication**: $a \equiv b \pmod{m} \Rightarrow ac \equiv bc \pmod{m}$
4. **Exponentiation**: $a \equiv b \pmod{m} \Rightarrow a^k \equiv b^k \pmod{m}$

Modular Inverse: $a^{-1} \bmod m$ exists iff $\gcd(a, m) = 1$

Found using Extended Euclidean Algorithm: If $ax + my = 1$, then $a^{-1} \equiv x \pmod{m}$

Chinese Remainder Theorem (CRT):

Theorem: System of congruences

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_k \pmod{m_k} \end{aligned}$$

has unique solution modulo $M = m_1 m_2 \cdots m_k$ if $\gcd(m_i, m_j) = 1$ for $i \neq j$

Construction:

1. Let $M_i = M/m_i$
2. Find y_i such that $M_i y_i \equiv 1 \pmod{m_i}$ (using Extended GCD)
3. Solution: $x \equiv \sum_{i=1}^{k} a_i M_i y_i \pmod{M}$

Example: Solve $x \equiv 2 \pmod{3}, x \equiv 3 \pmod{5}, x \equiv 2 \pmod{7}$

- $M = 3 \times 5 \times 7 = 105$
- $M_1 = 35, M_2 = 21, M_3 = 15$
- $35y_1 \equiv 1 \pmod{3} \Rightarrow 2y_1 \equiv 1 \pmod{3} \Rightarrow y_1 = 2$
- $21y_2 \equiv 1 \pmod{5} \Rightarrow 1y_2 \equiv 1 \pmod{5} \Rightarrow y_2 = 1$
- $15y_3 \equiv 1 \pmod{7} \Rightarrow 1y_3 \equiv 1 \pmod{7} \Rightarrow y_3 = 1$
- $x \equiv 2(35)(2) + 3(21)(1) + 2(15)(1) = 140 + 63 + 30 = 233 \equiv 23 \pmod{105}$

Euler's Totient Function:

Definition: $\phi(n)$ = number of integers in $\{1, 2, \ldots, n\}$ coprime to n

Formula: For $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$:

$$\phi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$

Properties:

1. $\phi(p) = p - 1$ for prime p
2. $\phi(p^k) = p^k - p^{k-1} = p^{k-1}(p - 1)$
3. **Multiplicative**: If $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m)\phi(n)$

4. $\sum_{d|n} \phi(d) = n$

Euler's Theorem: If $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$

Applications in Cryptography:

RSA Algorithm:

1. Choose primes p, q; compute $n = pq$
2. Choose e with $\gcd(e, \phi(n)) = 1$
3. Compute $d \equiv e^{-1} \pmod{\phi(n)}$
4. Public key: (n, e); Private key: (n, d)
5. Encrypt: $c \equiv m^e \pmod{n}$
6. Decrypt: $m \equiv c^d \pmod{n}$

Security: Based on difficulty of factoring large integers

Problem-Solving Strategy:

For GCD problems:

1. Use Euclidean algorithm for computation
2. Use Extended GCD for linear combinations
3. Apply properties: $\gcd(a, b) = \gcd(a \bmod b, b)$

For modular arithmetic:

1. Reduce numbers modulo m early and often
2. Use properties of congruence
3. For inverses, check $\gcd(a, m) = 1$ first

For prime problems:

1. Use fundamental theorem for factorization
2. Apply Fermat's Little Theorem for modular exponentiation
3. Use sieve methods for finding multiple primes

GATE Tips:

- Euclidean algorithm: $\gcd(a, b) = \gcd(b, a \bmod b)$
- Extended GCD gives Bézout coefficients: $ax + by = \gcd(a, b)$
- Modular inverse exists iff $\gcd(a, m) = 1$
- Chinese Remainder Theorem requires pairwise coprime moduli
- Fermat's Little Theorem: $a^{p-1} \equiv 1 \pmod{p}$ for prime p, $\gcd(a, p) = 1$
- Euler's theorem generalizes Fermat: $a^{\phi(n)} \equiv 1 \pmod{n}$ for $\gcd(a, n) = 1$
- $\phi(p^k) = p^{k-1}(p - 1)$ for prime p

- Fundamental theorem: Every integer has unique prime factorization
- For large numbers, use fast exponentiation: $a^{2k} = (a^k)^2$

Common GATE Examples:

Example 1: $\gcd(252, 198)$

```
252 = 198 × 1 + 54
198 = 54 × 3 + 36
54 = 36 × 1 + 18
36 = 18 × 2 + 0
```

Answer: $\gcd(252, 198) = 18$

Example 2: Find x such that $7x \equiv 1 \pmod{26}$
Using Extended GCD: $7(-11) + 26(3) = 1$
So $7(-11) \equiv 1 \pmod{26}$, thus $x \equiv -11 \equiv 15 \pmod{26}$

Example 3: Compute $3^{100} \bmod 7$
By Fermat: $3^6 \equiv 1 \pmod{7}$
$100 = 6 \times 16 + 4$
So $3^{100} = (3^6)^{16} \times 3^4 \equiv 1^{16} \times 81 \equiv 81 \equiv 4 \pmod{7}$

4.9 Onto (1) {#49-onto-1}

Key Concepts: Surjective: Every codomain hit. **Formulas/Theorems**: $|textimf| = |textcodomain|$. **Problem-Solving Tips**: Count preimages.

4.10 Partial Order (10) {#410-partial-order-10}

Key Concepts: Partial orders formalize the notion of "less than or equal to" relationships that may not compare all pairs of elements. Fundamental in discrete mathematics, computer science, and order theory.

Partial Order Definition:

Definition: A **partial order** (or **partial ordering relation**) on set S is a binary relation $\leq$ (or $\preceq$) that is:

1. **Reflexive**: $\forall a \in S : a \leq a$
2. **Antisymmetric**: $\forall a, b \in S : (a \leq b \land b \leq a) \Rightarrow a = b$
3. **Transitive**: $\forall a, b, c \in S : (a \leq b \land b \leq c) \Rightarrow a \leq c$

Partially Ordered Set (Poset): $(S, \leq)$ where $\leq$ is partial order on S

Strict Partial Order: Relation $<$ where $a < b \Leftrightarrow a \leq b \land a \neq b$

- **Irreflexive**: $a \not< a$

- **Asymmetric**: $a < b \Rightarrow b \not< a$
- **Transitive**: $a < b \wedge b < c \Rightarrow a < c$

Comparability:

Comparable Elements: $a, b \in S$ are **comparable** if $a \leq b$ or $b \leq a$

Incomparable Elements: a, b are **incomparable** (written $a \parallel b$) if neither $a \leq b$ nor $b \leq a$

Total Order (Linear Order): Partial order where every pair is comparable

- $\forall a, b \in S : a \leq b \text{ or } b \leq a$
- Examples: $\mathbb{R}, \mathbb{Z}, \mathbb{N}$ with usual $\leq$

Examples of Partial Orders:

1. Subset Relation $\subseteq$ on power set $\mathcal{P}(S)$:

- Reflexive: $A \subseteq A$
- Antisymmetric: $A \subseteq B \wedge B \subseteq A \Rightarrow A = B$
- Transitive: $A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C$
- Not total unless $|S| \leq 1$

2. Divisibility $|$ on positive integers:

- $a \mid b$ means "a divides b"
- Reflexive: $a \mid a$
- Antisymmetric: $a \mid b \wedge b \mid a \Rightarrow a = b$ (for positive integers)
- Transitive: $a \mid b \wedge b \mid c \Rightarrow a \mid c$
- Not total: 2 and 3 are incomparable

3. Lexicographic Order on strings:

- $(a_1, a_2, \ldots) \leq (b_1, b_2, \ldots)$ if:
  - $a_1 < b_1$, OR
  - $a_1 = b_1$ and $(a_2, a_3, \ldots) \leq (b_2, b_3, \ldots)$
- Total order on strings over totally ordered alphabet

4. Coordinate-wise Order on $\mathbb{R}^n$:

- $(a_1, \ldots, a_n) \leq (b_1, \ldots, b_n)$ if $a_i \leq b_i$ for all i
- Partial order (not total for $n > 1$)
- $(1, 2) \parallel (2, 1)$ in $\mathbb{R}^2$

5. Prefix Order on strings:

- $s \leq t$ if s is prefix of t

- Tree structure: each string has unique "parent"

Special Elements:

Maximal Element: a is **maximal** if $\nexists b \in S : a < b$

- No element strictly greater than a
- May have multiple maximal elements

Maximum Element: a is **maximum** (or **greatest**) if $\forall b \in S : b \leq a$

- Unique if exists
- Maximum element is maximal, but not vice versa

Minimal Element: a is **minimal** if $\nexists b \in S : b < a$

Minimum Element: a is **minimum** (or **least**) if $\forall b \in S : a \leq b$

Upper Bound: For subset $T \subseteq S$, element $u \in S$ is **upper bound** if $\forall t \in T : t \leq u$

Least Upper Bound (LUB/Supremum): Upper bound u such that for any upper bound v: $u \leq v$

- Notation: $\sup(T)$ or $\bigvee T$
- Unique if exists

Lower Bound: Element $l \in S$ is **lower bound** of T if $\forall t \in T : l \leq t$

Greatest Lower Bound (GLB/Infimum): Lower bound l such that for any lower bound m: $m \leq l$

- Notation: $\inf(T)$ or $\bigwedge T$

Chains and Antichains:

Chain: Subset $C \subseteq S$ where every pair is comparable

- Totally ordered subset
- Example: $\{1, 2, 6, 12\}$ in divisibility poset

Antichain: Subset $A \subseteq S$ where every pair is incomparable

- No two elements comparable
- Example: $\{6, 10, 15\}$ in divisibility poset on divisors of 30

Chain Decomposition: Partition of poset into disjoint chains

Antichain Decomposition: Partition of poset into disjoint antichains

Dilworth's Theorem:

Theorem: In any finite poset, the maximum size of an antichain equals the minimum number of chains needed to cover the poset.

Dual: The maximum size of a chain equals the minimum number of antichains needed to cover the poset.

Proof Idea: Use induction on poset size. Remove maximal and minimal elements, apply induction, then reconstruct.

Applications:

- **Scheduling**: Tasks with precedence constraints
- **Resource allocation**: Minimize parallel resources
- **Combinatorial optimization**: Matching theory

Width and Height:

Width: Size of largest antichain

- Measures "maximum parallelism"

Height: Size of largest chain

- Measures "maximum depth"

Hasse Diagrams:

Definition: Graph representation of finite poset

- Vertices: elements of poset
- Edges: **cover relations** only
- a **covers** b (written $b \lessdot a$) if $b < a$ and $\nexists c : b < c < a$

Drawing Rules:

1. Place smaller elements below larger ones
2. Draw edge only for cover relations
3. Omit reflexive loops and transitive edges
4. Transitivity: path upward means $\leq$ relation

Example: Divisors of 12

```
    \ /
     1
```

Reading Hasse Diagrams:

- $a \leq b$: Path from a upward to b
- Maximal elements: No outgoing edges (top)
- Minimal elements: No incoming edges (bottom)
- Chains: Paths in diagram
- Antichains: Sets with no path between any pair

Linear Extensions:

Definition: **Linear extension** of poset $(S, \leq)$ is total order $\leq'$ on S such that:
$a \leq b \Rightarrow a \leq' b$

Existence: Every finite poset has at least one linear extension

Topological Sorting: Algorithm to find linear extension

1. Find element with no predecessors
2. Remove it and add to sorted list
3. Repeat until all elements processed

Applications: Task scheduling with dependencies

Order-Preserving Functions:

Monotone Function: $f : (S_1, \leq_1) \to (S_2, \leq_2)$ is **monotone** (or **order-preserving**) if:
$a \leq_1 b \Rightarrow f(a) \leq_2 f(b)$

Order Isomorphism: Bijective function f where both f and f^{-1} are monotone

- Preserves order structure completely
- Isomorphic posets have same "shape"

Order Embedding: Injective monotone function f where:
$a \leq_1 b \Leftrightarrow f(a) \leq_2 f(b)$

Well-Ordered Sets:

Definition: Poset where every non-empty subset has minimum element

Properties:

- Total order
- No infinite descending chains
- Examples: $\mathbb{N}$, any finite total order
- $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ are NOT well-ordered

Well-Ordering Principle: Every non-empty subset of natural numbers has minimum element

- Equivalent to mathematical induction
- Foundation for transfinite induction

Zorn's Lemma: If every chain in poset has upper bound, then poset has maximal element

- Equivalent to Axiom of Choice
- Used in algebra (existence of bases, maximal ideals)

Applications in Computer Science:

1. Program Analysis:

- Control flow: Dominance relation
- Data flow: Information ordering
- Type systems: Subtyping hierarchy

2. Concurrency:

- Happened-before relation
- Causal ordering of events
- Vector clocks

3. Databases:

- Functional dependencies
- Query optimization
- Preference queries

4. Algorithms:

- Topological sorting
- Scheduling with precedence
- Partial order reduction

5. Formal Methods:

- Specification refinement
- Process algebra
- Model checking

Problem-Solving Strategy:

To verify partial order:

1. Check reflexivity: $a \leq a$ for all a

2. Check antisymmetry: $a \leq b \wedge b \leq a \Rightarrow a = b$
3. Check transitivity: $a \leq b \wedge b \leq c \Rightarrow a \leq c$

To find chains/antichains:

1. **Chain**: Find path in Hasse diagram
2. **Antichain**: Find elements with no order relation
3. **Maximum**: Use Dilworth's theorem

To construct Hasse diagram:

1. List all elements
2. Find cover relations (direct connections)
3. Draw with smaller elements below
4. Verify transitivity by paths

GATE Tips:

- Partial order: reflexive, antisymmetric, transitive
- Total order: partial order where all pairs comparable
- Hasse diagram: cover relations only, smaller below larger
- Chain: totally ordered subset (path in Hasse diagram)
- Antichain: pairwise incomparable subset
- Dilworth: max antichain size = min chain cover
- Well-order: every subset has minimum
- Topological sort: linear extension of partial order
- Maximal ≠ maximum (maximal may not be unique)

Common GATE Examples:

Example 1: Divisibility on {1,2,3,4,6,12}

- Partial order: $a \mid b$
- Hasse diagram: 1 at bottom, 12 at top
- Chain: {1,2,4,12} or {1,3,6,12}
- Antichain: {4,6} or {2,3}
- Width: 2, Height: 4

Example 2: Subset relation on $\mathcal{P}(\{a, b\})$

- Elements: $\emptyset, \{a\}, \{b\}, \{a, b\}$
- Chain: $\emptyset \subset \{a\} \subset \{a, b\}$
- Antichain: $\{\{a\}, \{b\}\}$
- Boolean lattice structure

Example 3: Coordinate-wise on $\{(1,1),(1,2),(2,1),(2,2)\}$

- $(1,1) \leq (1,2)$ and $(1,1) \leq (2,1)$
- $(1,2) \parallel (2,1)$ (incomparable)
- $(2,2)$ is maximum element
- Width: 2, Height: 3

4.11 Polynomials (4) {#411-polynomials-4}

Key Concepts: Polynomials are algebraic expressions with variables and coefficients. Fundamental in algebra, analysis, and computational mathematics.

Polynomial Definition:

Definition: A **polynomial** over field F (usually $\mathbb{R}$ or $\mathbb{C}$) is expression:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

where $a_i \in F$ are **coefficients** and $a_n \neq 0$

Degree: $\deg(p) = n$ (highest power with non-zero coefficient)

- Degree of zero polynomial: $-\infty$ or undefined
- Constant non-zero polynomial: degree 0

Leading Coefficient: a_n (coefficient of highest degree term)

Monic Polynomial: Leading coefficient is 1

Polynomial Operations:

Addition: $(p+q)(x) = p(x) + q(x)$

- $\deg(p+q) \leq \max(\deg(p), \deg(q))$

Multiplication: $(p \cdot q)(x) = p(x) \cdot q(x)$

- $\deg(p \cdot q) = \deg(p) + \deg(q)$
- Coefficient of x^k: $\sum_{i+j=k} a_i b_j$

Composition: $(p \circ q)(x) = p(q(x))$

- $\deg(p \circ q) = \deg(p) \cdot \deg(q)$

Division Algorithm for Polynomials:

Theorem: For polynomials $f(x), g(x)$ with $g(x) \neq 0$, there exist unique polynomials $q(x)$ (quotient) and $r(x)$ (remainder) such that:

$$f(x) = g(x) \cdot q(x) + r(x)$$

where $\deg(r) < \deg(g)$ or $r(x) = 0$

Long Division: Algorithm similar to integer division

Synthetic Division: Efficient method when dividing by $(x - c)$

Roots and Factors:

Root (Zero): Value r where $p(r) = 0$

Factor Theorem:

Theorem: $(x - r)$ is factor of $p(x)$ if and only if $p(r) = 0$

Proof ($\Rightarrow$): If $(x - r) \mid p(x)$, then $p(x) = (x - r)q(x)$
So $p(r) = (r - r)q(r) = 0$

Proof ($\Leftarrow$): By division algorithm: $p(x) = (x - r)q(x) + c$ where c is constant
Since $p(r) = 0$: $0 = 0 \cdot q(r) + c$, so $c = 0$
Therefore $(x - r) \mid p(x)$

Remainder Theorem:

Theorem: When $p(x)$ is divided by $(x - r)$, remainder is $p(r)$

Proof: $p(x) = (x - r)q(x) + c$ where c is constant
Substitute $x = r$: $p(r) = 0 + c = c$

Multiplicity:

Definition: Root r has **multiplicity** m if $(x - r)^m \mid p(x)$ but $(x - r)^{m+1} \nmid p(x)$

Properties:

- Simple root: multiplicity 1
- Multiple root: multiplicity > 1
- If r has multiplicity m, then $p(r) = p'(r) = \cdots = p^{(m-1)}(r) = 0$ but $p^{(m)}(r) \neq 0$

Fundamental Theorem of Algebra:

Theorem: Every non-constant polynomial with complex coefficients has at least one complex root.

Corollary: Polynomial of degree n has exactly n roots (counting multiplicities) in $\mathbb{C}$

Factorization: Over $\mathbb{C}$:

$$p(x) = a_n(x - r_1)(x - r_2)\cdots(x - r_n)$$

where r_i are roots (possibly repeated)

Real Polynomials:

Complex Conjugate Root Theorem: If $p(x)$ has real coefficients and $a + bi$ is root, then $a - bi$ is also root

Factorization over $\mathbb{R}$: Product of linear and irreducible quadratic factors:

$$p(x) = a_n(x - r_1) \cdots (x - r_k)(x^2 + b_1x + c_1) \cdots (x^2 + b_mx + c_m)$$

where quadratics have no real roots ($b_i^2 - 4c_i < 0$)

Finding Roots:

Rational Root Theorem:

Theorem: If $p(x) = a_nx^n + \cdots + a_0$ has integer coefficients and $\frac{p}{q}$ is rational root (in lowest terms), then:

- p divides a_0 (constant term)
- q divides a_n (leading coefficient)

Strategy: Test all candidates $\pm\frac{p}{q}$ where $p \mid a_0$ and $q \mid a_n$

Example: $p(x) = 2x^3 - 3x^2 - 11x + 6$

- Possible rational roots: $\pm 1, \pm 2, \pm 3, \pm 6, \pm\frac{1}{2}, \pm\frac{3}{2}$
- Test: $p(3) = 54 - 27 - 33 + 6 = 0$ ✓
- Factor: $p(x) = (x - 3)(2x^2 + 3x - 2) = (x - 3)(2x - 1)(x + 2)$

Descartes' Rule of Signs:

Theorem: Number of positive real roots of $p(x)$ is either equal to number of sign changes in coefficients, or less by even number.

Similarly for negative roots using $p(-x)$.

Quadratic Formula: For $ax^2 + bx + c = 0$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Discriminant: $\Delta = b^2 - 4ac$

- $\Delta > 0$: Two distinct real roots
- $\Delta = 0$: One repeated real root
- $\Delta < 0$: Two complex conjugate roots

Cubic Formula: Exists but complicated (Cardano's formula)

Quartic Formula: Exists but very complicated (Ferrari's method)

Abel-Ruffini Theorem: No general algebraic formula for roots of polynomials of degree ≥ 5

Vieta's Formulas:

For polynomial $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ with roots $r_1, \ldots, r_n$:

$$r_1 + r_2 + \cdots + r_n = -\frac{a_{n-1}}{a_n}$$

$$r_1 r_2 + r_1 r_3 + \cdots + r_{n-1} r_n = \frac{a_{n-2}}{a_n}$$

$$\vdots$$

$$r_1 r_2 \cdots r_n = (-1)^n \frac{a_0}{a_n}$$

For quadratic $ax^2 + bx + c$:

- Sum of roots: $r_1 + r_2 = -\frac{b}{a}$
- Product of roots: $r_1 r_2 = \frac{c}{a}$

Polynomial Interpolation:

Lagrange Interpolation: Given $n+1$ points $(x_0, y_0), \ldots, (x_n, y_n)$ with distinct x_i, unique polynomial of degree $\leq n$ passes through all points:

$$p(x) = \sum_{i=0}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Newton's Divided Differences: Alternative interpolation method

Irreducibility:

Definition: Polynomial $p(x)$ over field F is **irreducible** if it cannot be factored into product of non-constant polynomials over F

Over $\mathbb{C}$: Only linear polynomials are irreducible (by Fundamental Theorem)

Over $\mathbb{R}$: Linear and quadratics with negative discriminant are irreducible

Over $\mathbb{Q}$: More complex

Eisenstein's Criterion: If $p(x) = a_n x^n + \cdots + a_0$ with integer coefficients and there exists prime p such that:

1. $p \nmid a_n$
2. $p \mid a_i$ for all $i < n$
3. $p^2 \nmid a_0$

Then $p(x)$ is irreducible over $\mathbb{Q}$

Example: $p(x) = x^4 + 2x^3 + 2x^2 + 2x + 2$
Using $p = 2$: $2 \nmid 1$, $2 \mid 2$, $4 \nmid 2$ ✓ Irreducible

Applications in Computer Science:

1. Error-Correcting Codes:

- Reed-Solomon codes use polynomial evaluation
- BCH codes use minimal polynomials

2. Cryptography:

- Finite field arithmetic (AES uses $GF(2^8)$)
- Polynomial-based key exchange

3. Computer Graphics:

- Bézier curves (Bernstein polynomials)
- Spline interpolation

4. Numerical Analysis:

- Root finding (Newton's method)
- Polynomial approximation

5. Symbolic Computation:

- Computer algebra systems
- Gröbner bases

Problem-Solving Strategy:

To find roots:

1. Try rational root theorem for integer/rational roots
2. Use quadratic formula for degree 2
3. Factor by grouping or substitution
4. Numerical methods for higher degrees

To factor:

1. Find one root using rational root theorem
2. Use synthetic division to reduce degree
3. Repeat until fully factored

To check irreducibility:

1. Try Eisenstein's criterion

2. Check for rational roots
3. For quadratics: check discriminant

GATE Tips:

- Factor theorem: $(x - r)$ divides $p(x)$ iff $p(r) = 0$
- Remainder theorem: Remainder when dividing by $(x - r)$ is $p(r)$
- Rational root theorem: Test $\pm \frac{\text{factors of } a_0}{\text{factors of } a_n}$
- Degree of product: $\deg(pq) = \deg(p) + \deg(q)$
- Polynomial of degree n has at most n roots
- Vieta's formulas relate roots to coefficients
- Complex roots come in conjugate pairs for real polynomials
- Quadratic discriminant: $b^2 - 4ac$

Common GATE Examples:

Example 1: Factor $x^3 - 6x^2 + 11x - 6$

- Rational roots: $\pm 1, \pm 2, \pm 3, \pm 6$
- Test: $p(1) = 1 - 6 + 11 - 6 = 0$ ✓
- Divide: $(x - 1)(x^2 - 5x + 6) = (x - 1)(x - 2)(x - 3)$

Example 2: Find sum and product of roots of $2x^2 - 5x + 3 = 0$

- Sum: $-\frac{-5}{2} = \frac{5}{2}$
- Product: $\frac{3}{2}$
- Verify: Roots are $\frac{3}{2}, 1$; sum = $\frac{5}{2}$, product = $\frac{3}{2}$ ✓

Example 3: Check if $x^2 + 1$ irreducible over $\mathbb{R}$

- Discriminant: $0 - 4(1)(1) = -4 < 0$
- No real roots, so irreducible over $\mathbb{R}$
- But factors over $\mathbb{C}$: $(x - i)(x + i)$

4.12 Relations (37) {#412-relations-37}

Key Concepts: Relations formalize connections between elements of sets. Fundamental for databases, discrete mathematics, and computer science applications.

Relation Definition:

Binary Relation: A **binary relation** R from set A to set B is subset $R \subseteq A \times B$

- Write aRb or $(a, b) \in R$ to mean "a is related to b"
- **Domain**: $\text{dom}(R) = \{a \in A : \exists b \in B, (a, b) \in R\}$
- **Range**: $\text{range}(R) = \{b \in B : \exists a \in A, (a, b) \in R\}$

Relation on Set: When $A = B$, $R \subseteq A \times A$ is **relation on** A

Matrix Representation: For finite sets $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_n\}$:

$$M_R[i,j] = \begin{cases} 1 & \text{if } (a_i, b_j) \in R \\ 0 & \text{otherwise} \end{cases}$$

Directed Graph Representation:

- Vertices: elements of A
- Edges: $(a,b) \in R$ represented as arrow from a to b

Properties of Relations on Set A:

1. Reflexive: $\forall a \in A : (a,a) \in R$

- Every element related to itself
- Matrix: All diagonal entries are 1
- Graph: Self-loop at every vertex
- Example: $\leq$ on $\mathbb{R}$

2. Irreflexive: $\forall a \in A : (a,a) \notin R$

- No element related to itself
- Matrix: All diagonal entries are 0
- Graph: No self-loops
- Example: $<$ on $\mathbb{R}$

3. Symmetric: $\forall a,b \in A : (a,b) \in R \Rightarrow (b,a) \in R$

- If a related to b, then b related to a
- Matrix: $M_R = M_R^T$ (symmetric matrix)
- Graph: Edges are bidirectional
- Example: "is sibling of" relation

4. Antisymmetric: $\forall a,b \in A : [(a,b) \in R \wedge (b,a) \in R] \Rightarrow a = b$

- If both (a,b) and (b,a) in relation, then $a = b$
- Matrix: If $M_R[i,j] = 1$ and $i \neq j$, then $M_R[j,i] = 0$
- Example: $\leq$ on $\mathbb{R}$

5. Asymmetric: $\forall a,b \in A : (a,b) \in R \Rightarrow (b,a) \notin R$

- Stronger than antisymmetric: no bidirectional edges
- Implies irreflexive
- Example: $<$ on $\mathbb{R}$

6. Transitive: $\forall a,b,c \in A : [(a,b) \in R \wedge (b,c) \in R] \Rightarrow (a,c) \in R$

- If a related to b and b related to c, then a related to c
- Matrix: If path of length 2 exists, direct edge must exist
- Example: $\leq$ on $\mathbb{R}$

Special Types of Relations:

Equivalence Relation:

Definition: Relation that is **reflexive**, **symmetric**, and **transitive**

Examples:

- Equality ($=$) on any set
- Congruence modulo n: $a \equiv b \pmod{n}$
- "Same birthday" on set of people
- "Similar triangles" in geometry

Equivalence Classes:

Definition: For equivalence relation $\sim$ on A and element $a \in A$:

$$[a] = \{x \in A : x \sim a\}$$

Properties:

1. $a \in [a]$ (non-empty)
2. $[a] = [b] \Leftrightarrow a \sim b$
3. $[a] \cap [b] = \emptyset$ or $[a] = [b]$ (disjoint or identical)
4. $\bigcup_{a \in A} [a] = A$ (cover entire set)

Partition: Collection of non-empty, pairwise disjoint subsets whose union is entire set

Fundamental Theorem: Equivalence relations on A correspond bijectively to partitions of A

- Given equivalence relation → equivalence classes form partition
- Given partition → define equivalence by "in same block"

Quotient Set: $A/\sim = \{[a] : a \in A\}$ (set of all equivalence classes)

Partial Order Relation:

Definition: Relation that is **reflexive**, **antisymmetric**, and **transitive**

Examples:

- $\leq$ on $\mathbb{R}$
- $\subseteq$ on power set
- Divisibility $|$ on positive integers
- Prefix relation on strings

Strict Partial Order: **Irreflexive**, **asymmetric**, **transitive**

- Example: $<$ on $\mathbb{R}$
- Connection: $a < b \Leftrightarrow a \leq b \wedge a \neq b$

Total Order (Linear Order): Partial order where every pair is comparable

- $\forall a, b \in A : a \leq b \text{ or } b \leq a$
- Examples: $\leq$ on $\mathbb{R}$, lexicographic order on strings

Operations on Relations:

Union: $R_1 \cup R_2 = \{(a, b) : (a, b) \in R_1 \text{ or } (a, b) \in R_2\}$

Intersection: $R_1 \cap R_2 = \{(a, b) : (a, b) \in R_1 \text{ and } (a, b) \in R_2\}$

Complement: $\overline{R} = (A \times A) \setminus R$

Inverse (Converse): $R^{-1} = \{(b, a) : (a, b) \in R\}$

- Matrix: $M_{R^{-1}} = M_R^T$
- Properties:
  - $(R^{-1})^{-1} = R$
  - R symmetric $\Leftrightarrow R = R^{-1}$

Composition: For relations $R_1 : A \to B$ and $R_2 : B \to C$:

$$R_2 \circ R_1 = \{(a, c) : \exists b \in B, (a, b) \in R_1 \wedge (b, c) \in R_2\}$$

Matrix Composition: $M_{R_2 \circ R_1} = M_{R_1} \cdot M_{R_2}$ (Boolean matrix multiplication)

- $(M_1 \cdot M_2)[i, j] = \bigvee_k (M_1[i, k] \wedge M_2[k, j])$

Properties of Composition:

- **Associative**: $(R_3 \circ R_2) \circ R_1 = R_3 \circ (R_2 \circ R_1)$
- **Not commutative**: Generally $R_2 \circ R_1 \neq R_1 \circ R_2$
- **Identity**: $I_A \circ R = R \circ I_B = R$ where I is identity relation

Powers of Relation: $R^n = R \circ R \circ \cdots \circ R$ (n times)

- $R^0 = I_A$ (identity relation)
- $R^1 = R$
- $R^{n+1} = R^n \circ R$

Closures of Relations:

Reflexive Closure: $r(R) = R \cup I_A$

- Smallest reflexive relation containing R
- Add all pairs (a, a) not already in R

Symmetric Closure: $s(R) = R \cup R^{-1}$

- Smallest symmetric relation containing R
- Add (b, a) whenever $(a, b) \in R$

Transitive Closure: $t(R) = R^1 \cup R^2 \cup R^3 \cup \cdots$

- Smallest transitive relation containing R
- $(a, c) \in t(R)$ iff there exists path from a to c in R

Warshall's Algorithm (Computing Transitive Closure):

```
Warshall(M):  // M is adjacency matrix of R
    n = size of matrix
    for k = 1 to n:
        for i = 1 to n:
            for j = 1 to n:
                M[i][j] = M[i][j] OR (M[i][k] AND M[k][j])
    return M
```

Time Complexity: $O(n^3)$

Idea: $M^{(k)}[i, j] = 1$ iff path from i to j using only vertices $\{1, 2, \ldots, k\}$ as intermediates

Floyd-Warshall: Similar algorithm for shortest paths in weighted graphs

Equivalence Closure: $e(R) = rst(R)$ (reflexive, symmetric, transitive closure)

- Can apply closures in any order
- Results in equivalence relation

Applications in Computer Science:

1. Databases:

- **Relational model**: Tables as relations
- **Functional dependencies**: $X \to Y$ in database design
- **Join operations**: Composition of relations
- **Normalization**: Based on functional dependencies

2. Program Analysis:

- **Control flow**: Relations between program points
- **Data dependencies**: Variable usage relations
- **Alias analysis**: "May point to" relations

3. Formal Methods:

- **State transitions**: Relations between system states
- **Refinement**: Relations between specifications
- **Bisimulation**: Equivalence relations on processes

4. Graph Theory:

- **Adjacency relation**: Edge relation in graphs
- **Reachability**: Transitive closure of adjacency
- **Strongly connected components**: Equivalence classes

5. Type Systems:

- **Subtyping**: Partial order on types
- **Type equivalence**: Equivalence relation on types

Problem-Solving Strategy:

To check relation properties:

1. **Reflexive**: Check if $(a, a) \in R$ for all a
2. **Symmetric**: Check if $(a, b) \in R \Rightarrow (b, a) \in R$
3. **Transitive**: Check if $(a, b), (b, c) \in R \Rightarrow (a, c) \in R$
4. **Antisymmetric**: Check if $(a, b), (b, a) \in R \Rightarrow a = b$

To find equivalence classes:

1. Start with any element a
2. Find all elements related to a
3. This forms equivalence class $[a]$
4. Repeat with unprocessed elements

To compute transitive closure:

1. **Small relations**: Add paths manually
2. **Matrix form**: Use Warshall's algorithm
3. **Graph form**: Use DFS/BFS from each vertex

GATE Tips:

- Equivalence relation: reflexive + symmetric + transitive
- Partial order: reflexive + antisymmetric + transitive
- Equivalence classes partition the set
- Transitive closure: union of all powers $R^1 \cup R^2 \cup \cdots$
- Warshall's algorithm: $O(n^3)$ for transitive closure

- Matrix composition: Boolean matrix multiplication
- Symmetric relation: $R = R^{-1}$
- Function is special case of relation (each input has unique output)

Common GATE Examples:

Example 1: Check if $R = \{(1,1), (2,2), (3,3), (1,2), (2,1)\}$ on $\{1,2,3\}$ is equivalence

- Reflexive: $(1,1), (2,2), (3,3) \in R$ ✓
- Symmetric: $(1,2) \in R$ and $(2,1) \in R$ ✓
- Transitive: Need $(1,1), (2,2)$ from $(1,2), (2,1)$ - already present ✓
- **Answer**: Yes, equivalence relation
- **Equivalence classes**: $\{\{1,2\}, \{3\}\}$

Example 2: Find transitive closure of $R = \{(1,2), (2,3), (3,1)\}$

- $R^1 = \{(1,2), (2,3), (3,1)\}$
- $R^2 = \{(1,3), (2,1), (3,2)\}$ (composition with self)
- $R^3 = \{(1,1), (2,2), (3,3)\}$
- $R^4 = R^1$ (cycle repeats)
- **Transitive closure**:
  $R^1 \cup R^2 \cup R^3 = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}$

Example 3: Matrix representation of relation $R = \{(1,2), (2,1), (2,3)\}$ on $\{1,2,3\}$

$$M_R = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Example 4: Composition $R_1 \circ R_2$ where:

- $R_1 = \{(1,a), (2,b)\}$
- $R_2 = \{(a,x), (b,y), (b,z)\}$
- $R_1 \circ R_2 = \{(1,x), (2,y), (2,z)\}$

4.13 Set Theory (27) {#413-set-theory-27}

Key Concepts: Set theory provides the foundation for modern mathematics. Sets are collections of distinct objects, forming the basis for defining numbers, functions, and mathematical structures.

Basic Definitions:

Set: Well-defined collection of distinct objects called **elements** or **members**

- Notation: $A = \{1,2,3\}$ or $A = \{x : P(x)\}$ (set-builder notation)
- **Element of**: $a \in A$ means "a is an element of A"

- **Not element of**: $a \notin A$ means "a is not an element of A"

Empty Set: $\emptyset = \{\}$ (set with no elements)

- Unique and subset of every set
- $|\emptyset| = 0$

Universal Set: U contains all objects under consideration in given context

Cardinality: $|A|$ denotes number of elements in finite set A

Set Equality: $A = B$ iff A and B have exactly same elements

- **Extensionality**: Sets determined entirely by their elements
- Order doesn't matter: $\{1, 2, 3\} = \{3, 1, 2\}$
- Repetition doesn't matter: $\{1, 1, 2\} = \{1, 2\}$

Subset Relations:

Subset: $A \subseteq B$ iff every element of A is also element of B

- Formal: $A \subseteq B \Leftrightarrow \forall x(x \in A \Rightarrow x \in B)$
- **Reflexive**: $A \subseteq A$
- **Transitive**: $A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C$
- **Antisymmetric**: $A \subseteq B \wedge B \subseteq A \Rightarrow A = B$

Proper Subset: $A \subset B$ (or $A \subsetneq B$) iff $A \subseteq B$ and $A \neq B$

- Equivalently: $A \subseteq B$ and $\exists x \in B$ such that $x \notin A$

Properties:

- $\emptyset \subseteq A$ for any set A
- $A \subseteq U$ for any set A (if U is universal set)
- $A \subseteq B$ and $B \subseteq A$ iff $A = B$

Set Operations:

Union: $A \cup B = \{x : x \in A \text{ or } x \in B\}$

- Elements in either set (or both)
- **Commutative**: $A \cup B = B \cup A$
- **Associative**: $(A \cup B) \cup C = A \cup (B \cup C)$
- **Idempotent**: $A \cup A = A$
- **Identity**: $A \cup \emptyset = A$
- **Domination**: $A \cup U = U$

Intersection: $A \cap B = \{x : x \in A \text{ and } x \in B\}$

- Elements in both sets
- **Commutative**: $A \cap B = B \cap A$
- **Associative**: $(A \cap B) \cap C = A \cap (B \cap C)$
- **Idempotent**: $A \cap A = A$
- **Identity**: $A \cap U = A$
- **Domination**: $A \cap \emptyset = \emptyset$

Complement: $A^c = \overline{A} = U \setminus A = \{x \in U : x \notin A\}$

- Elements in universal set but not in A
- **Involution**: $(A^c)^c = A$
- **Complement laws**: $A \cup A^c = U$, $A \cap A^c = \emptyset$
- **Universal laws**: $U^c = \emptyset$, $\emptyset^c = U$

Set Difference: $A \setminus B = A - B = \{x : x \in A \text{ and } x \notin B\}$

- Elements in A but not in B
- **Not commutative**: Generally $A \setminus B \neq B \setminus A$
- **Relationship to complement**: $A^c = U \setminus A$

Symmetric Difference: $A \triangle B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$

- Elements in exactly one of the sets
- **Commutative**: $A \triangle B = B \triangle A$
- **Associative**: $(A \triangle B) \triangle C = A \triangle (B \triangle C)$
- **Identity**: $A \triangle \emptyset = A$
- **Self-inverse**: $A \triangle A = \emptyset$

Fundamental Laws of Set Theory:

Distributive Laws:

- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
- $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

De Morgan's Laws:

- $(A \cup B)^c = A^c \cap B^c$
- $(A \cap B)^c = A^c \cup B^c$

Proof of De Morgan's Law: $(A \cup B)^c = A^c \cap B^c$

$x \in (A \cup B)^c$
$\Leftrightarrow x \notin (A \cup B)$
$\Leftrightarrow x \notin A \text{ and } x \notin B$
$\Leftrightarrow x \in A^c \text{ and } x \in B^c$
$\Leftrightarrow x \in A^c \cap B^c$

Absorption Laws:

- $A \cup (A \cap B) = A$
- $A \cap (A \cup B) = A$

Generalized De Morgan's Laws:

- $\left(\bigcup_{i \in I} A_i\right)^c = \bigcap_{i \in I} A_i^c$
- $\left(\bigcap_{i \in I} A_i\right)^c = \bigcup_{i \in I} A_i^c$

Power Set:

Definition: $\mathcal{P}(A) = 2^A = \{X : X \subseteq A\}$ (set of all subsets of A)

Properties:

- $\emptyset \in \mathcal{P}(A)$ and $A \in \mathcal{P}(A)$
- $|\mathcal{P}(A)| = 2^{|A|}$ for finite set A
- $\mathcal{P}(\emptyset) = \{\emptyset\}$ (not empty!)
- $\mathcal{P}(\{a\}) = \{\emptyset, \{a\}\}$
- $\mathcal{P}(\{a, b\}) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

Proof of $|\mathcal{P}(A)| = 2^{|A|}$:
For each subset $S \subseteq A$, define characteristic function $\chi_S : A \to \{0, 1\}$:

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

Bijection between subsets of A and functions $A \to \{0, 1\}$
Number of such functions = $2^{|A|}$

Cartesian Product:

Definition: $A \times B = \{(a, b) : a \in A \text{ and } b \in B\}$

- Set of all ordered pairs
- **Order matters**: $(a, b) \neq (b, a)$ unless $a = b$

Properties:

- $|A \times B| = |A| \cdot |B|$
- **Not commutative**: $A \times B \neq B \times A$ (unless $A = B$)
- **Not associative**: $(A \times B) \times C \neq A \times (B \times C)$ (different structures)
- $A \times \emptyset = \emptyset \times A = \emptyset$

Distributive Laws for Cartesian Product:

- $A \times (B \cup C) = (A \times B) \cup (A \times C)$

- $A \times (B \cap C) = (A \times B) \cap (A \times C)$
- $(A \cup B) \times C = (A \times C) \cup (B \times C)$
- $(A \cap B) \times C = (A \times C) \cap (B \times C)$

n-fold Cartesian Product: $A^n = A \times A \times \cdots \times A$ (n times)

- $A^0 = \{()\}$ (set containing empty tuple)
- $A^1 = A$ (identified with A)
- $|A^n| = |A|^n$

Principle of Inclusion-Exclusion:

For finite sets:

Two sets: $|A \cup B| = |A| + |B| - |A \cap B|$

Three sets: $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$

General formula: For sets $A_1, A_2, \ldots, A_n$:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i} |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n-1} |A_1 \cap A_2 \cap \cdots \cap A_n|$$

Venn Diagrams:

Purpose: Visual representation of set relationships

Two sets: Two overlapping circles

- 4 regions: $A \cap B^c$, $A \cap B$, $A^c \cap B$, $A^c \cap B^c$

Three sets: Three overlapping circles

- 8 regions representing all possible intersections
- Useful for verifying set identities

Limitations: Difficult for more than 3 sets

Disjoint Sets:

Definition: Sets A and B are **disjoint** if $A \cap B = \emptyset$

Pairwise Disjoint: Collection $\{A_i\}_{i \in I}$ is pairwise disjoint if $A_i \cap A_j = \emptyset$ for all $i \neq j$

Partition: Collection of non-empty, pairwise disjoint sets whose union is entire set

- Formal: $\{A_i\}_{i \in I}$ is partition of A if:
  1. $A_i \neq \emptyset$ for all i
  2. $A_i \cap A_j = \emptyset$ for $i \neq j$

3. $\bigcup_{i \in I} A_i = A$

Russell's Paradox:

Naive Set Theory Problem: Consider set $R = \{x : x \notin x\}$ (set of all sets that don't contain themselves)

Question: Is $R \in R$?

- If $R \in R$, then by definition of R, $R \notin R$ (contradiction)
- If $R \notin R$, then by definition of R, $R \in R$ (contradiction)

Resolution: Led to axiomatic set theory (ZFC - Zermelo-Fraenkel with Choice)

Applications in Computer Science:

1. Data Structures:

- Sets as abstract data type
- Hash sets, tree sets
- Set operations in programming languages

2. Databases:

- Relational algebra based on set theory
- Union, intersection, difference operations
- Set-based query languages

3. Formal Methods:

- Specification languages (Z, VDM)
- Set-theoretic semantics
- Model checking

4. Complexity Theory:

- Complexity classes as sets of problems
- Set relationships (P ⊆ NP)

5. Logic and AI:

- Knowledge representation
- Semantic networks
- Description logics

Problem-Solving Strategy:

To prove set equality $A = B$:

1. **Method 1**: Show $A \subseteq B$ and $B \subseteq A$
2. **Method 2**: Show $x \in A \Leftrightarrow x \in B$ for arbitrary x
3. **Method 3**: Use known set identities

To find cardinality:

1. **Direct counting**: List elements
2. **Inclusion-exclusion**: For unions
3. **Bijection**: Show correspondence with known set
4. **Recursive**: Break into smaller cases

To verify set identity:

1. **Algebraic**: Use set laws
2. **Element-wise**: Show membership equivalence
3. **Venn diagrams**: Visual verification (up to 3 sets)
4. **Counterexample**: Find case where identity fails

GATE Tips:

- De Morgan's laws: $(A \cup B)^c = A^c \cap B^c$, $(A \cap B)^c = A^c \cup B^c$
- Power set cardinality: $|\mathcal{P}(A)| = 2^{|A|}$
- Inclusion-exclusion: $|A \cup B| = |A| + |B| - |A \cap B|$
- Distributive laws work both ways for ∪ and ∩
- Cartesian product: $|A \times B| = |A| \cdot |B|$
- Empty set is subset of every set: $\emptyset \subseteq A$
- Set difference not commutative: $A \setminus B \neq B \setminus A$
- Symmetric difference: $A \triangle B = (A \cup B) \setminus (A \cap B)$

Common GATE Examples:

Example 1: Verify $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$?
Answer: False. Counterexample: $A = \{1\}, B = \{2\}, C = \{3\}$

- LHS: $(\{1\} \cap \{2\}) \cup \{3\} = \emptyset \cup \{3\} = \{3\}$
- RHS: $(\{1\} \cup \{3\}) \cap (\{2\} \cup \{3\}) = \{1,3\} \cap \{2,3\} = \{3\}$
- Actually, this IS true! Correct identity.

Example 2: If $|A| = 5, |B| = 3, |A \cap B| = 2$, find $|A \cup B|$

- By inclusion-exclusion: $|A \cup B| = |A| + |B| - |A \cap B| = 5 + 3 - 2 = 6$

Example 3: Find $\mathcal{P}(\{a, \{a\}\})$

- Elements of given set: a and $\{a\}$ (two distinct elements)
- Subsets: $\emptyset, \{a\}, \{\{a\}\}, \{a, \{a\}\}$

- Answer: $\mathcal{P}(\{a, \{a\}\}) = \{\emptyset, \{a\}, \{\{a\}\}, \{a, \{a\}\}\}$

Example 4: Simplify $(A \cup B^c) \cap (A^c \cup B)$

- Distribute: $(A \cap A^c) \cup (A \cap B) \cup (B^c \cap A^c) \cup (B^c \cap B)$
- Simplify: $\emptyset \cup (A \cap B) \cup (A^c \cap B^c) \cup \emptyset$
- Result: $(A \cap B) \cup (A^c \cap B^c) = (A \leftrightarrow B)$ (elements where A and B have same membership)

5. Engineering Mathematics: Calculus (150 Questions)

5.1 Limits and Continuity (25) {#51-limits-and-continuity-25}

Key Concepts: Limits form the foundation of calculus, formalizing the notion of "approaching" a value. Continuity describes functions without breaks or jumps.

Limit Definition:

Informal Definition: $\lim_{x \to a} f(x) = L$ means $f(x)$ gets arbitrarily close to L as x approaches a

Formal Definition (ε-δ): $\lim_{x \to a} f(x) = L$ if:

$$\forall \varepsilon > 0, \exists \delta > 0 \text{ such that } 0 < |x - a| < \delta \Rightarrow |f(x) - L| < \varepsilon$$

Geometric Interpretation: For any horizontal strip of width 2ε around L, there exists vertical strip of width 2δ around a such that graph lies within horizontal strip

One-Sided Limits:

Right-hand limit: $\lim_{x \to a^+} f(x) = L$ if limit exists as x approaches a from right
Left-hand limit: $\lim_{x \to a^-} f(x) = L$ if limit exists as x approaches a from left

Theorem: $\lim_{x \to a} f(x) = L$ iff $\lim_{x \to a^+} f(x) = \lim_{x \to a^-} f(x) = L$

Limit Laws (Assuming limits exist):

1. **Sum Rule**: $\lim_{x \to a}[f(x) + g(x)] = \lim_{x \to a} f(x) + \lim_{x \to a} g(x)$
2. **Product Rule**: $\lim_{x \to a}[f(x) \cdot g(x)] = \lim_{x \to a} f(x) \cdot \lim_{x \to a} g(x)$
3. **Quotient Rule**: $\lim_{x \to a} \frac{f(x)}{g(x)} = \frac{\lim_{x \to a} f(x)}{\lim_{x \to a} g(x)}$ (if denominator limit ≠ 0)
4. **Power Rule**: $\lim_{x \to a}[f(x)]^n = [\lim_{x \to a} f(x)]^n$
5. **Root Rule**: $\lim_{x \to a} \sqrt[n]{f(x)} = \sqrt[n]{\lim_{x \to a} f(x)}$ (if root is defined)

Standard Limits:

$$\lim_{x \to 0} \frac{\sin x}{x} = 1$$

Proof: Use squeeze theorem with $\cos x \leq \frac{\sin x}{x} \leq 1$ for $0 < |x| < \frac{\pi}{2}$

$$\lim_{x \to 0} \frac{1 - \cos x}{x^2} = \frac{1}{2}$$

$$\lim_{x \to 0} (1 + x)^{1/x} = e$$

$$\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x = e$$

$$\lim_{x \to 0} \frac{e^x - 1}{x} = 1$$

$$\lim_{x \to 0} \frac{\ln(1 + x)}{x} = 1$$

Indeterminate Forms:

Forms that require special techniques:

- $\frac{0}{0}$, $\frac{\infty}{\infty}$, $0 \cdot \infty$, $\infty - \infty$, 0^0, 1^∞, ∞^0

L'Hôpital's Rule: For indeterminate forms $\frac{0}{0}$ or $\frac{\infty}{\infty}$:

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \lim_{x \to a} \frac{f'(x)}{g'(x)}$$

provided the right limit exists

Conditions:

1. $\lim_{x \to a} f(x) = \lim_{x \to a} g(x) = 0$ or $\pm\infty$
2. $f'(x)$ and $g'(x)$ exist in neighborhood of a
3. $g'(x) \neq 0$ in neighborhood of a
4. $\lim_{x \to a} \frac{f'(x)}{g'(x)}$ exists

Squeeze Theorem (Sandwich Theorem):

If $g(x) \leq f(x) \leq h(x)$ for all x in neighborhood of a and $\lim_{x \to a} g(x) = \lim_{x \to a} h(x) = L$, then $\lim_{x \to a} f(x) = L$

Application: Proving $\lim_{x \to 0} x \sin(1/x) = 0$ using $-|x| \leq x \sin(1/x) \leq |x|$

Limits at Infinity:

$$\lim_{x \to \infty} f(x) = L \text{ means } \forall \varepsilon > 0, \exists M > 0 \text{ such that } x > M \Rightarrow |f(x) - L| < \varepsilon$$

Rational Functions: For $f(x) = \frac{a_n x^n + \cdots + a_0}{b_m x^m + \cdots + b_0}$:

$$\lim_{x \to \infty} f(x) = \begin{cases} 0 & \text{if } n < m \\ \frac{a_n}{b_m} & \text{if } n = m \\ \pm\infty & \text{if } n > m \end{cases}$$

Continuity:

Definition: Function f is **continuous at** a if:

1. $f(a)$ is defined
2. $\lim_{x \to a} f(x)$ exists
3. $\lim_{x \to a} f(x) = f(a)$

Equivalent: $\lim_{x \to a} f(x) = f(a)$

Types of Discontinuities:

1. **Removable**: $\lim_{x \to a} f(x)$ exists but $f(a)$ undefined or $f(a) \neq \lim_{x \to a} f(x)$
   - Can be "fixed" by redefining $f(a)$
2. **Jump**: Left and right limits exist but are unequal
   - $\lim_{x \to a^-} f(x) \neq \lim_{x \to a^+} f(x)$
3. **Infinite**: At least one one-sided limit is infinite
4. **Oscillating**: Limit doesn't exist due to oscillation (e.g., $\sin(1/x)$ at $x = 0$)

Properties of Continuous Functions:

Theorem: If f and g are continuous at a, then:

- $f + g$, $f - g$, $f \cdot g$ are continuous at a
- f/g is continuous at a if $g(a) \neq 0$
- $f \circ g$ is continuous at a if g continuous at a and f continuous at $g(a)$

Intermediate Value Theorem (IVT):

Theorem: If f is continuous on $[a, b]$ and k is between $f(a)$ and $f(b)$, then $\exists c \in (a, b)$ such that $f(c) = k$

Applications:

- Proving existence of roots
- Bisection method for root finding
- Fixed point theorems

Extreme Value Theorem:

Theorem: If f is continuous on closed interval $[a, b]$, then f attains its maximum and minimum values

Uniform Continuity:

Definition: f is **uniformly continuous** on interval I if:

$$\forall \varepsilon > 0, \exists \delta > 0 \text{ such that } |x - y| < \delta \Rightarrow |f(x) - f(y)| < \varepsilon$$

for all $x, y \in I$

Key Difference: Same δ works for all points (not dependent on specific point)

Theorem: Every continuous function on closed interval is uniformly continuous

Problem-Solving Techniques:

For Limits:

1. **Direct substitution**: If function continuous at point
2. **Factoring**: Cancel common factors for $\frac{0}{0}$ forms
3. **Rationalization**: Multiply by conjugate for radical expressions
4. **L'Hôpital's rule**: For indeterminate forms
5. **Squeeze theorem**: When function bounded between two others
6. **Standard limits**: Memorize key trigonometric and exponential limits

For Continuity:

1. **Check definition**: Verify $\lim_{x \to a} f(x) = f(a)$
2. **Piecewise functions**: Check continuity at boundary points
3. **Composition**: Use continuity of component functions

GATE Tips:

- L'Hôpital's rule only for $\frac{0}{0}$ or $\frac{\infty}{\infty}$ forms
- Standard limit $\lim_{x \to 0} \frac{\sin x}{x} = 1$ is fundamental
- IVT guarantees existence, not uniqueness
- Continuous on closed interval $\implies$ bounded and attains extrema
- For piecewise functions, check left and right limits at boundaries

Examples:

1. **Evaluate** $\lim_{x \to 0} \frac{\sin 3x}{x}$:

$$\lim_{x \to 0} \frac{\sin 3x}{x} = \lim_{x \to 0} \frac{\sin 3x}{3x} \cdot 3 = 1 \cdot 3 = 3$$

2. **Find** $\lim_{x \to 1} \frac{x^2 - 1}{x - 1}$:

$$\lim_{x \to 1} \frac{x^2 - 1}{x - 1} = \lim_{x \to 1} \frac{(x - 1)(x + 1)}{x - 1} = \lim_{x \to 1} (x + 1) = 2$$

3. **Check continuity** of $f(x) = \begin{cases} x^2 & x < 1 \\ 2x & x \geq 1 \end{cases}$ at $x = 1$:
   - $f(1) = 2(1) = 2$
   - $\lim_{x \to 1^-} f(x) = \lim_{x \to 1^-} x^2 = 1$
   - $\lim_{x \to 1^+} f(x) = \lim_{x \to 1^+} 2x = 2$
   - Since $\lim_{x \to 1^-} f(x) \neq \lim_{x \to 1^+} f(x)$, function has jump discontinuity at $x = 1$

5.2 Differentiation (40)

Key Concepts: Differentiation measures instantaneous rate of change. The derivative is the limit of difference quotients, providing slope of tangent line and velocity interpretation.

Definition of Derivative:

Limit Definition:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Alternative Form:

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}$$

Geometric Interpretation: Slope of tangent line to curve $y = f(x)$ at point $(a, f(a))$

Physical Interpretation: Instantaneous rate of change (velocity if f represents position)

Notation:

- $f'(x)$, $\frac{df}{dx}$, $\frac{d}{dx}f(x)$, $Df(x)$

Differentiability and Continuity:

Theorem: If f is differentiable at a, then f is continuous at a

Proof:

$$\lim_{x \to a}[f(x) - f(a)] = \lim_{x \to a} \frac{f(x) - f(a)}{x - a} \cdot (x - a) = f'(a) \cdot 0 = 0$$

Converse is False: Continuity doesn't imply differentiability

- Example: $f(x) = |x|$ at $x = 0$

Non-differentiable Points:

1. **Corner**: Left and right derivatives exist but are unequal
2. **Cusp**: Tangent line is vertical
3. **Vertical tangent**: Derivative is infinite
4. **Discontinuity**: Function not continuous

Basic Differentiation Rules:

Constant Rule: $\frac{d}{dx}[c] = 0$

Power Rule: $\frac{d}{dx}[x^n] = nx^{n-1}$ (for any real n)

Proof for positive integer n (using binomial theorem):

$$\frac{d}{dx}[x^n] = \lim_{h \to 0} \frac{(x+h)^n - x^n}{h} = \lim_{h \to 0} \frac{\sum_{k=1}^{n} \binom{n}{k} x^{n-k} h^k}{h} = nx^{n-1}$$

Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = c\frac{d}{dx}[f(x)]$

Sum Rule: $\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$

Product Rule: $\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$

Proof:

$$\frac{d}{dx}[f(x)g(x)] = \lim_{h \to 0} \frac{f(x+h)g(x+h) - f(x)g(x)}{h}$$

$$= \lim_{h \to 0} \frac{f(x+h)g(x+h) - f(x)g(x+h) + f(x)g(x+h) - f(x)g(x)}{h}$$

$$= \lim_{h \to 0} \left[g(x+h)\frac{f(x+h) - f(x)}{h} + f(x)\frac{g(x+h) - g(x)}{h} \right]$$

$$= g(x)f'(x) + f(x)g'(x)$$

Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{f'(x)g(x) - f(x)g'(x)}{[g(x)]^2}$

Memory Aid: "Low dee-high minus high dee-low, over low squared"

Chain Rule: $\frac{d}{dx}[f(g(x))] = f'(g(x)) \cdot g'(x)$

Leibniz Notation: $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$ where $y = f(u)$ and $u = g(x)$

Proof: Let $y = f(u)$ and $u = g(x)$

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta u} \cdot \frac{\Delta u}{\Delta x}$$

Derivatives of Elementary Functions:

Trigonometric Functions:

- $\frac{d}{dx}[\sin x] = \cos x$
- $\frac{d}{dx}[\cos x] = -\sin x$
- $\frac{d}{dx}[\tan x] = \sec^2 x$
- $\frac{d}{dx}[\cot x] = -\csc^2 x$
- $\frac{d}{dx}[\sec x] = \sec x \tan x$
- $\frac{d}{dx}[\csc x] = -\csc x \cot x$

Proof of $\frac{d}{dx}[\sin x] = \cos x$:

$$\frac{d}{dx}[\sin x] = \lim_{h \to 0} \frac{\sin(x+h) - \sin x}{h}$$

$$= \lim_{h \to 0} \frac{\sin x \cos h + \cos x \sin h - \sin x}{h}$$

$$= \lim_{h \to 0} \left[\sin x \frac{\cos h - 1}{h} + \cos x \frac{\sin h}{h} \right]$$

$$= \sin x \cdot 0 + \cos x \cdot 1 = \cos x$$

Exponential and Logarithmic Functions:

- $\frac{d}{dx}[e^x] = e^x$
- $\frac{d}{dx}[a^x] = a^x \ln a$
- $\frac{d}{dx}[\ln x] = \frac{1}{x}$
- $\frac{d}{dx}[\log_a x] = \frac{1}{x \ln a}$

Inverse Trigonometric Functions:

- $\frac{d}{dx}[\sin^{-1} x] = \frac{1}{\sqrt{1-x^2}}$
- $\frac{d}{dx}[\cos^{-1} x] = -\frac{1}{\sqrt{1-x^2}}$
- $\frac{d}{dx}[\tan^{-1} x] = \frac{1}{1+x^2}$

Hyperbolic Functions:

- $\frac{d}{dx}[\sinh x] = \cosh x$
- $\frac{d}{dx}[\cosh x] = \sinh x$
- $\frac{d}{dx}[\tanh x] = \operatorname{sech}^2 x$

Implicit Differentiation:

For equations not solved for y, differentiate both sides with respect to x, treating y as function of x

Example: Find $\frac{dy}{dx}$ for $x^2 + y^2 = 25$

$$\frac{d}{dx}[x^2 + y^2] = \frac{d}{dx}[25]$$

$$2x + 2y\frac{dy}{dx} = 0$$

$$\frac{dy}{dx} = -\frac{x}{y}$$

Logarithmic Differentiation:

For functions of form $y = [f(x)]^{g(x)}$ or products/quotients of many terms:

1. Take natural logarithm: $\ln y = g(x) \ln f(x)$
2. Differentiate implicitly: $\frac{1}{y}\frac{dy}{dx} = g'(x)\ln f(x) + g(x)\frac{f'(x)}{f(x)}$
3. Solve for $\frac{dy}{dx}$: $\frac{dy}{dx} = y[g'(x)\ln f(x) + g(x)\frac{f'(x)}{f(x)}]$

Example: $y = x^x$

$$\ln y = x \ln x$$

$$\frac{1}{y}\frac{dy}{dx} = \ln x + x \cdot \frac{1}{x} = \ln x + 1$$

$$\frac{dy}{dx} = x^x(\ln x + 1)$$

Higher-Order Derivatives:

Notation:

- Second derivative: $f''(x)$, $\frac{d^2f}{dx^2}$, $\frac{d^2y}{dx^2}$
- n-th derivative: $f^{(n)}(x)$, $\frac{d^nf}{dx^n}$

Leibniz Rule (Product rule for higher derivatives):

$$\frac{d^n}{dx^n}[f(x)g(x)] = \sum_{k=0}^{n} \binom{n}{k} f^{(k)}(x)g^{(n-k)}(x)$$

Parametric Differentiation:

For parametric equations $x = f(t)$, $y = g(t)$:

$$\frac{dy}{dx} = \frac{dy/dt}{dx/dt} = \frac{g'(t)}{f'(t)}$$

Second derivative:

$$\frac{d^2y}{dx^2} = \frac{d}{dx}\left(\frac{dy}{dx}\right) = \frac{d}{dt}\left(\frac{dy}{dx}\right) \cdot \frac{dt}{dx} = \frac{\frac{d}{dt}\left(\frac{dy}{dx}\right)}{dx/dt}$$

Related Rates:

Problems involving rates of change of related quantities

Strategy:

1. Identify variables and given rates
2. Find equation relating variables
3. Differentiate with respect to time
4. Substitute known values and solve

Example: Balloon inflating at 10 cm³/s. Find rate of radius change when $r = 5$ cm

- Volume: $V = \frac{4}{3}\pi r^3$
- Given: $\frac{dV}{dt} = 10$
- Find: $\frac{dr}{dt}$ when $r = 5$
- Differentiate: $\frac{dV}{dt} = 4\pi r^2 \frac{dr}{dt}$
- Substitute: $10 = 4\pi(25)\frac{dr}{dt}$
- Solve: $\frac{dr}{dt} = \frac{10}{100\pi} = \frac{1}{10\pi}$ cm/s

Applications of Derivatives:

Tangent and Normal Lines:

- Tangent line at $(a, f(a))$: $y - f(a) = f'(a)(x - a)$
- Normal line: $y - f(a) = -\frac{1}{f'(a)}(x - a)$ (if $f'(a) \neq 0$)

Linear Approximation:

$$f(x) \approx f(a) + f'(a)(x - a)$$

for x near a

Differential: $dy = f'(x)dx$ approximates $\Delta y = f(x + \Delta x) - f(x)$

Mean Value Theorem (MVT):

Theorem: If f is continuous on $[a, b]$ and differentiable on (a, b), then $\exists c \in (a, b)$ such that:

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

Geometric Interpretation: There exists point where tangent line is parallel to secant line

Rolle's Theorem (Special case of MVT):
If f continuous on $[a, b]$, differentiable on (a, b), and $f(a) = f(b)$, then $\exists c \in (a, b)$ such that $f'(c) = 0$

Problem-Solving Tips:

For Basic Differentiation:

1. Identify which rules apply (product, quotient, chain)
2. Work from outside in for composite functions
3. Simplify before differentiating when possible

For Implicit Differentiation:

1. Differentiate both sides term by term
2. Remember to multiply by $\frac{dy}{dx}$ when differentiating y terms
3. Collect all $\frac{dy}{dx}$ terms and solve

For Related Rates:

1. Draw diagram if geometric
2. Identify all variables and their relationships
3. Differentiate the constraint equation
4. Substitute known values at specific instant

GATE Tips:

- Power rule works for any real exponent: $\frac{d}{dx}[x^r] = rx^{r-1}$
- Chain rule is essential: always identify inner and outer functions
- Product rule: $(fg)' = f'g + fg'$ (not $f'g'$!)
- For $\frac{0}{0}$ limits, try L'Hôpital's rule after checking conditions
- Implicit differentiation: treat y as function of x
- Related rates: differentiate constraint equation with respect to time
- MVT guarantees existence of point, not uniqueness

Examples:

1. **Find** $\frac{d}{dx}[(x^2+1)^3 \sin x]$:
   Using product rule and chain rule:

$$\frac{d}{dx}[(x^2+1)^3 \sin x] = 3(x^2+1)^2 \cdot 2x \cdot \sin x + (x^2+1)^3 \cos x$$

$$= 6x(x^2+1)^2 \sin x + (x^2+1)^3 \cos x$$

2. **Find** $\frac{dy}{dx}$ for $x^3 + y^3 = 6xy$:
   Differentiating implicitly:

$$3x^2 + 3y^2 \frac{dy}{dx} = 6y + 6x \frac{dy}{dx}$$

$$3y^2 \frac{dy}{dx} - 6x \frac{dy}{dx} = 6y - 3x^2$$

$$\frac{dy}{dx} = \frac{6y - 3x^2}{3y^2 - 6x} = \frac{2y - x^2}{y^2 - 2x}$$

3. **Verify MVT** for $f(x) = x^2$ on $[1, 3]$:
   - $f'(x) = 2x$
   - $\frac{f(3)-f(1)}{3-1} = \frac{9-1}{2} = 4$
   - Need $f'(c) = 4$: $2c = 4 \Rightarrow c = 2$
   - Since $2 \in (1, 3)$, MVT is satisfied with $c = 2$

5.3 Integration (50)

Key Concepts: Integration is the reverse process of differentiation, used to find areas, volumes, and accumulated quantities. The Fundamental Theorem of Calculus connects differentiation and integration.

Antiderivative Definition:

Definition: Function $F(x)$ is an **antiderivative** of $f(x)$ if $F'(x) = f(x)$

General Antiderivative: $F(x) + C$ where C is arbitrary constant

Indefinite Integral: $\int f(x)dx = F(x) + C$ where $F'(x) = f(x)$

Definite Integral:

Riemann Sum Definition:

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\Delta x$$

where $\Delta x = \frac{b-a}{n}$ and $x_i^* \in [x_{i-1}, x_i]$

Geometric Interpretation: Signed area between curve and x-axis from $x = a$ to $x = b$

Properties of Definite Integrals:

1. **Linearity**: $\int_a^b [cf(x) + dg(x)]dx = c\int_a^b f(x)dx + d\int_a^b g(x)dx$
2. **Additivity**: $\int_a^b f(x)dx + \int_b^c f(x)dx = \int_a^c f(x)dx$
3. **Reversal**: $\int_a^b f(x)dx = -\int_b^a f(x)dx$
4. **Zero width**: $\int_a^a f(x)dx = 0$
5. **Comparison**: If $f(x) \leq g(x)$ on $[a, b]$, then $\int_a^b f(x)dx \leq \int_a^b g(x)dx$

Fundamental Theorem of Calculus:

Part I (Evaluation Theorem):
If f is continuous on $[a, b]$ and F is antiderivative of f, then:

$$\int_a^b f(x)dx = F(b) - F(a) = [F(x)]_a^b$$

Part II (Derivative of Integral):
If f is continuous on $[a, b]$, then:

$$\frac{d}{dx}\int_a^x f(t)dt = f(x)$$

More General Form:

$$\frac{d}{dx}\int_{g(x)}^{h(x)} f(t)dt = f(h(x))h'(x) - f(g(x))g'(x)$$

Basic Integration Formulas:

Power Rule: $\int x^n dx = \frac{x^{n+1}}{n+1} + C$ (for $n \neq -1$)

Logarithmic: $\int \frac{1}{x}dx = \ln|x| + C$

Exponential:

- $\int e^x dx = e^x + C$
- $\int a^x dx = \frac{a^x}{\ln a} + C$

Trigonometric:

- $\int \sin x dx = -\cos x + C$
- $\int \cos x dx = \sin x + C$
- $\int \sec^2 x dx = \tan x + C$
- $\int \csc^2 x dx = -\cot x + C$
- $\int \sec x \tan x dx = \sec x + C$
- $\int \csc x \cot x dx = -\csc x + C$

Inverse Trigonometric:

- $\int \frac{1}{\sqrt{1-x^2}} dx = \sin^{-1} x + C$
- $\int \frac{1}{1+x^2} dx = \tan^{-1} x + C$
- $\int \frac{1}{x\sqrt{x^2-1}} dx = \sec^{-1} |x| + C$

Integration Techniques:

1. Substitution Method (u-substitution):

Indefinite: If $\int f(g(x))g'(x)dx$, let $u = g(x)$, $du = g'(x)dx$

$$\int f(g(x))g'(x)dx = \int f(u)du$$

Definite: $\int_a^b f(g(x))g'(x)dx = \int_{g(a)}^{g(b)} f(u)du$

Example: $\int 2x(x^2+1)^3 dx$

- Let $u = x^2 + 1$, $du = 2xdx$
- $\int (x^2+1)^3 \cdot 2xdx = \int u^3 du = \frac{u^4}{4} + C = \frac{(x^2+1)^4}{4} + C$

2. Integration by Parts:

Formula: $\int udv = uv - \int vdu$

Choosing u and dv (LIATE priority):

- **L**ogarithmic functions
- **I**nverse trigonometric functions
- **A**lgebraic functions (polynomials)
- **T**rigonometric functions
- **E**xponential functions

Example: $\int xe^x dx$

- Let $u = x$, $dv = e^x dx$
- Then $du = dx$, $v = e^x$
- $\int xe^x dx = xe^x - \int e^x dx = xe^x - e^x + C = e^x(x-1) + C$

Repeated Integration by Parts:
For $\int x^n e^x dx$, apply parts n times

3. Trigonometric Integrals:

Powers of sine and cosine:

- $\int \sin^m x \cos^n x dx$ where m or n is odd: substitute for the even power
- Both even: use half-angle formulas

Half-angle formulas:

- $\sin^2 x = \frac{1-\cos 2x}{2}$
- $\cos^2 x = \frac{1+\cos 2x}{2}$

Powers of tangent and secant:

- $\int \tan^m x \sec^n x dx$ where n is even: substitute $u = \tan x$
- m is odd: substitute $u = \sec x$

4. Trigonometric Substitution:

For integrals involving:

- $\sqrt{a^2 - x^2}$: use $x = a \sin \theta$
- $\sqrt{a^2 + x^2}$: use $x = a \tan \theta$
- $\sqrt{x^2 - a^2}$: use $x = a \sec \theta$

Example: $\int \frac{1}{\sqrt{4-x^2}} dx$

- Let $x = 2 \sin \theta$, $dx = 2 \cos \theta d\theta$
- $\sqrt{4 - x^2} = \sqrt{4 - 4 \sin^2 \theta} = 2 \cos \theta$
- $\int \frac{2 \cos \theta}{2 \cos \theta} d\theta = \int d\theta = \theta + C = \sin^{-1} \frac{x}{2} + C$

5. Partial Fractions:

For rational functions $\frac{P(x)}{Q(x)}$ where degree of $P <$ degree of Q:

Linear factors: $\frac{A}{x-a} + \frac{B}{x-b} + \cdots$

Repeated linear factors: $\frac{A_1}{x-a} + \frac{A_2}{(x-a)^2} + \cdots + \frac{A_n}{(x-a)^n}$

Quadratic factors: $\frac{Ax+B}{x^2+px+q}$ (for irreducible quadratics)

Example: $\int \frac{x+1}{x^2-x-2} dx = \int \frac{x+1}{(x-2)(x+1)} dx$

- $\frac{x+1}{(x-2)(x+1)} = \frac{A}{x-2} + \frac{B}{x+1}$
- $x + 1 = A(x + 1) + B(x - 2)$

- Setting $x = 2$: $3 = 3A \Rightarrow A = 1$
- Setting $x = -1$: $0 = -3B \Rightarrow B = 0$
- $\int \frac{1}{x-2} dx = \ln|x-2| + C$

6. Numerical Integration:

Trapezoidal Rule:

$$\int_a^b f(x)dx \approx \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)]$$

where $h = \frac{b-a}{n}$

Simpson's Rule (n even):

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 4f(x_{n-1}) + f(x_n)]$$

Error Analysis:

- Trapezoidal: Error $\leq \frac{(b-a)^3}{12n^2} \max|f''(x)|$
- Simpson's: Error $\leq \frac{(b-a)^5}{180n^4} \max|f^{(4)}(x)|$

Improper Integrals:

Type I (Infinite limits):

$$\int_a^\infty f(x)dx = \lim_{t\to\infty} \int_a^t f(x)dx$$

Type II (Discontinuous integrand):

$$\int_a^b f(x)dx = \lim_{t\to c^-} \int_a^t f(x)dx + \lim_{t\to c^+} \int_t^b f(x)dx$$

where f has discontinuity at $x = c \in (a, b)$

Convergence Tests:

- **Comparison Test**: If $0 \leq f(x) \leq g(x)$ and $\int g(x)dx$ converges, then $\int f(x)dx$ converges
- **Limit Comparison Test**: If $\lim_{x\to\infty} \frac{f(x)}{g(x)} = L > 0$, then both integrals converge or both diverge

Applications of Integration:

Area between curves:

$$A = \int_a^b |f(x) - g(x)|dx$$

Volume by cross-sections:

$$V = \int_a^b A(x)dx$$

where $A(x)$ is area of cross-section at x

Volume of revolution:

- **Disk method**: $V = \pi \int_a^b [f(x)]^2 dx$
- **Washer method**: $V = \pi \int_a^b ([f(x)]^2 - [g(x)]^2) dx$
- **Shell method**: $V = 2\pi \int_a^b x f(x) dx$

Arc length:

$$L = \int_a^b \sqrt{1 + [f'(x)]^2} dx$$

Surface area of revolution:

$$S = 2\pi \int_a^b f(x) \sqrt{1 + [f'(x)]^2} dx$$

Problem-Solving Strategy:

For Integration:

1. **Identify type**: Basic formula, substitution, parts, trig substitution, partial fractions
2. **Simplify first**: Factor, expand, or rewrite if helpful
3. **Check answer**: Differentiate result to verify

For Definite Integrals:

1. **Find antiderivative** using appropriate technique
2. **Apply FTC**: Evaluate at bounds and subtract
3. **Check reasonableness**: Consider sign and magnitude

GATE Tips:

- Master basic formulas and substitution method first
- Integration by parts: choose u using LIATE
- Trig substitution: match radical form to substitution
- Partial fractions: degree of numerator < degree of denominator
- FTC Part II: $\frac{d}{dx} \int_a^x f(t)dt = f(x)$
- Improper integrals: check convergence before evaluating
- Area problems: determine which function is on top

Examples:

1. **Evaluate** $\int x^2 e^{x^3} dx$:

- Let $u = x^3$, $du = 3x^2 dx$, so $x^2 dx = \frac{1}{3} du$
- $\int x^2 e^{x^3} dx = \frac{1}{3} \int e^u du = \frac{1}{3} e^u + C = \frac{1}{3} e^{x^3} + C$

2. **Evaluate** $\int_0^{\pi/2} x \sin x dx$:
   - Using integration by parts: $u = x$, $dv = \sin x dx$
   - $du = dx$, $v = -\cos x$
   - $\int x \sin x dx = -x \cos x + \int \cos x dx = -x \cos x + \sin x + C$
   - $\int_0^{\pi/2} x \sin x dx = [-x \cos x + \sin x]_0^{\pi/2} = [0 + 1] - [0 + 0] = 1$
3. **Find area** between $y = x^2$ and $y = 2x$ from $x = 0$ to $x = 2$:
   - Intersection points: $x^2 = 2x \Rightarrow x = 0, 2$
   - For $0 \leq x \leq 2$: $2x \geq x^2$
   - $A = \int_0^2 (2x - x^2) dx = [x^2 - \frac{x^3}{3}]_0^2 = 4 - \frac{8}{3} = \frac{4}{3}$

5.4 Differential Equations (35)

Key Concepts: Differential equations involve functions and their derivatives. They model rates of change in physics, engineering, biology, and economics.

Classification of Differential Equations:

By Order: Highest derivative present

- **First-order**: $\frac{dy}{dx} = f(x, y)$
- **Second-order**: $\frac{d^2y}{dx^2} = f(x, y, y')$
- **n-th order**: Contains $\frac{d^n y}{dx^n}$

By Linearity:

- **Linear**: Dependent variable and derivatives appear linearly
- **Nonlinear**: Contains products, powers, or nonlinear functions of y and derivatives

By Homogeneity:

- **Homogeneous**: All terms involve dependent variable or its derivatives
- **Non-homogeneous**: Contains terms independent of dependent variable

First-Order Differential Equations:

1. Separable Equations:

Form: $\frac{dy}{dx} = f(x)g(y)$ or $M(x)dx + N(y)dy = 0$

Solution Method:

1. Separate variables: $\frac{dy}{g(y)} = f(x)dx$
2. Integrate both sides: $\int \frac{dy}{g(y)} = \int f(x)dx$
3. Solve for y if possible

Example: $\frac{dy}{dx} = xy$

- Separate: $\frac{dy}{y} = xdx$
- Integrate: $\ln|y| = \frac{x^2}{2} + C$
- Solve: $y = Ae^{x^2/2}$ where $A = \pm e^C$

2. Linear First-Order Equations:

Standard Form: $\frac{dy}{dx} + P(x)y = Q(x)$

Solution Method (Integrating Factor):

1. Find integrating factor: $\mu(x) = e^{\int P(x)dx}$
2. Multiply equation by $\mu(x)$: $\mu(x)\frac{dy}{dx} + \mu(x)P(x)y = \mu(x)Q(x)$
3. Left side becomes $\frac{d}{dx}[\mu(x)y]$
4. Integrate: $\mu(x)y = \int \mu(x)Q(x)dx$
5. Solve for y: $y = \frac{1}{\mu(x)}\int \mu(x)Q(x)dx$

Example: $\frac{dy}{dx} + 2y = e^{-x}$

- $P(x) = 2$, $Q(x) = e^{-x}$
- $\mu(x) = e^{\int 2dx} = e^{2x}$
- $e^{2x}\frac{dy}{dx} + 2e^{2x}y = e^{2x} \cdot e^{-x} = e^x$
- $\frac{d}{dx}[e^{2x}y] = e^x$
- $e^{2x}y = \int e^x dx = e^x + C$
- $y = e^{-x} + Ce^{-2x}$

3. Exact Equations:

Form: $M(x,y)dx + N(x,y)dy = 0$

Exactness Condition: $\frac{\partial M}{\partial y} = \frac{\partial N}{\partial x}$

Solution Method:

1. Check exactness condition
2. Find function $F(x,y)$ such that $\frac{\partial F}{\partial x} = M$ and $\frac{\partial F}{\partial y} = N$
3. Solution is $F(x,y) = C$

Finding F(x,y):

- $F(x,y) = \int M(x,y)dx + g(y)$
- Determine $g(y)$ using $\frac{\partial F}{\partial y} = N$

4. Homogeneous Equations:

Form: $\frac{dy}{dx} = f\left(\frac{y}{x}\right)$

Solution Method:

1. Substitute $v = \frac{y}{x}$, so $y = vx$ and $\frac{dy}{dx} = v + x\frac{dv}{dx}$
2. Equation becomes: $v + x\frac{dv}{dx} = f(v)$
3. Separate: $x\frac{dv}{dx} = f(v) - v$
4. $\frac{dv}{f(v)-v} = \frac{dx}{x}$
5. Integrate and substitute back

Second-Order Linear Differential Equations:

General Form: $a(x)\frac{d^2y}{dx^2} + b(x)\frac{dy}{dx} + c(x)y = f(x)$

Constant Coefficients: $ay'' + by' + cy = f(x)$

Homogeneous Case: $ay'' + by' + cy = 0$

Characteristic Equation Method:

1. Assume solution $y = e^{rx}$
2. Substitute: $ar^2 + br + c = 0$
3. Solve quadratic for r

Cases for Roots:

Case 1: Two distinct real roots r_1, r_2

- General solution: $y = c_1e^{r_1x} + c_2e^{r_2x}$

Case 2: Repeated real root r

- General solution: $y = (c_1 + c_2x)e^{rx}$

Case 3: Complex roots $r = \alpha \pm \beta i$

- General solution: $y = e^{\alpha x}(c_1 \cos \beta x + c_2 \sin \beta x)$

Example: $y'' - 5y' + 6y = 0$

- Characteristic equation: $r^2 - 5r + 6 = 0$
- Factor: $(r - 2)(r - 3) = 0$
- Roots: $r_1 = 2, r_2 = 3$
- Solution: $y = c_1e^{2x} + c_2e^{3x}$

Non-homogeneous Equations: $ay'' + by' + cy = f(x)$

General Solution: $y = y_h + y_p$

- y_h: homogeneous solution
- y_p: particular solution

Method of Undetermined Coefficients:

For specific forms of $f(x)$:

$f(x)$	**Trial** y_p
ae^{kx}	Ae^{kx}
$a\cos kx + b\sin kx$	$A\cos kx + B\sin kx$
ax^n	$A_nx^n + A_{n-1}x^{n-1} + \cdots + A_0$
$ae^{kx}\cos mx$	$e^{kx}(A\cos mx + B\sin mx)$

Modification Rule: If trial solution is part of homogeneous solution, multiply by x (or x^2 if necessary)

Example: $y'' - 3y' + 2y = e^x$

- Homogeneous: $r^2 - 3r + 2 = 0 \Rightarrow r = 1, 2$
- $y_h = c_1e^x + c_2e^{2x}$
- Since e^x is in y_h, try $y_p = Axe^x$
- $y_p' = A(e^x + xe^x) = Ae^x(1 + x)$
- $y_p'' = Ae^x(2 + x)$
- Substitute: $Ae^x(2 + x) - 3Ae^x(1 + x) + 2Axe^x = e^x$
- $Ae^x(2 + x - 3 - 3x + 2x) = e^x$
- $Ae^x(-1) = e^x \Rightarrow A = -1$
- $y_p = -xe^x$
- General solution: $y = c_1e^x + c_2e^{2x} - xe^x$

Variation of Parameters:

For $y'' + p(x)y' + q(x)y = f(x)$ with known homogeneous solutions y_1, y_2:

Particular Solution:

$$y_p = -y_1\int \frac{y_2f(x)}{W}dx + y_2\int \frac{y_1f(x)}{W}dx$$

where $W = y_1y_2' - y_2y_1'$ is the Wronskian

Applications of Differential Equations:

1. Population Growth:

- **Exponential**: $\frac{dP}{dt} = kP \Rightarrow P(t) = P_0e^{kt}$
- **Logistic**: $\frac{dP}{dt} = kP(1 - \frac{P}{M}) \Rightarrow P(t) = \frac{M}{1 + Ae^{-kt}}$

2. Newton's Law of Cooling:

$$\frac{dT}{dt} = -k(T - T_{\text{ambient}})$$

Solution: $T(t) = T_{\text{ambient}} + (T_0 - T_{\text{ambient}})e^{-kt}$

3. Simple Harmonic Motion:

$$m\frac{d^2x}{dt^2} + kx = 0$$

Solution: $x(t) = A\cos(\omega t + \phi)$ where $\omega = \sqrt{\frac{k}{m}}$

4. RLC Circuits:

$$L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{q}{C} = E(t)$$

5. Mixing Problems:
Rate of change = Rate in - Rate out

$$\frac{dA}{dt} = r_{\text{in}}c_{\text{in}} - r_{\text{out}}\frac{A(t)}{V(t)}$$

Laplace Transform Method:

Definition: $\mathcal{L}\{f(t)\} = F(s) = \int_0^\infty e^{-st}f(t)dt$

Key Properties:

- $\mathcal{L}\{f'(t)\} = sF(s) - f(0)$
- $\mathcal{L}\{f''(t)\} = s^2F(s) - sf(0) - f'(0)$
- $\mathcal{L}\{e^{at}f(t)\} = F(s-a)$

Common Transforms:

- $\mathcal{L}\{1\} = \frac{1}{s}$
- $\mathcal{L}\{t^n\} = \frac{n!}{s^{n+1}}$
- $\mathcal{L}\{e^{at}\} = \frac{1}{s-a}$
- $\mathcal{L}\{\sin at\} = \frac{a}{s^2+a^2}$
- $\mathcal{L}\{\cos at\} = \frac{s}{s^2+a^2}$

Solution Process:

1. Take Laplace transform of DE
2. Solve algebraic equation for $Y(s) = \mathcal{L}\{y(t)\}$
3. Find inverse transform: $y(t) = \mathcal{L}^{-1}\{Y(s)\}$

Problem-Solving Strategy:

For First-Order DEs:

1. **Identify type**: Separable, linear, exact, homogeneous
2. **Apply appropriate method**
3. **Include arbitrary constant**
4. **Apply initial conditions** if given

For Second-Order DEs:

1. **Find homogeneous solution** using characteristic equation
2. **Find particular solution** using undetermined coefficients or variation of parameters
3. **Combine**: $y = y_h + y_p$
4. **Apply initial conditions**

GATE Tips:

- Separable: $\frac{dy}{dx} = f(x)g(y) \rightarrow$ separate and integrate
- Linear first-order: Use integrating factor $\mu(x) = e^{\int P(x)dx}$
- Characteristic equation: $ar^2 + br + c = 0$ for $ay'' + by' + cy = 0$
- Complex roots $\alpha \pm \beta i$ give $e^{\alpha x}(c_1 \cos \beta x + c_2 \sin \beta x)$
- Undetermined coefficients: Match form of non-homogeneous term
- Initial conditions determine arbitrary constants
- Check solutions by substitution

Examples:

1. **Solve** $\frac{dy}{dx} = \frac{x}{y}$ with $y(0) = 1$:
   - Separate: $ydy = xdx$
   - Integrate: $\frac{y^2}{2} = \frac{x^2}{2} + C$
   - $y^2 = x^2 + 2C$
   - Initial condition: $1 = 0 + 2C \Rightarrow C = \frac{1}{2}$
   - Solution: $y^2 = x^2 + 1 \Rightarrow y = \sqrt{x^2 + 1}$ (taking positive root)
2. **Solve** $y'' + 4y = 0$:
   - Characteristic equation: $r^2 + 4 = 0$
   - Roots: $r = \pm 2i$
   - Solution: $y = c_1 \cos 2x + c_2 \sin 2x$
3. **Solve** $y'' - y = e^x$:
   - Homogeneous: $r^2 - 1 = 0 \Rightarrow r = \pm 1$
   - $y_h = c_1 e^x + c_2 e^{-x}$
   - Since e^x is in y_h, try $y_p = Axe^x$
   - $y_p' = A(e^x + xe^x)$, $y_p'' = A(2e^x + xe^x)$
   - Substitute: $A(2e^x + xe^x) - Axe^x = e^x$
   - $2Ae^x = e^x \Rightarrow A = \frac{1}{2}$
   - Solution: $y = c_1 e^x + c_2 e^{-x} + \frac{1}{2}xe^x$

6. Engineering Mathematics: Linear Algebra (120 Questions)

6.1 Matrices and Determinants (40)

Key Concepts: Matrices are rectangular arrays of numbers that represent linear transformations and systems of equations. Determinants measure how matrices scale areas and volumes.

Matrix Definitions:

Matrix: Rectangular array of numbers arranged in rows and columns

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Notation: $A = [a_{ij}]_{m \times n}$ where i is row index, j is column index

Special Matrices:

Square Matrix: $m = n$ (same number of rows and columns)

Zero Matrix: All entries are zero, denoted O or 0

Identity Matrix: Square matrix with 1's on diagonal, 0's elsewhere

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

Diagonal Matrix: Square matrix with non-zero entries only on main diagonal

$$D = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix}$$

Upper Triangular: $a_{ij} = 0$ for $i > j$
Lower Triangular: $a_{ij} = 0$ for $i < j$

Symmetric Matrix: $A = A^T$ (equals its transpose)
Skew-Symmetric: $A = -A^T$

Matrix Operations:

Matrix Addition: $(A + B)_{ij} = a_{ij} + b_{ij}$

- Only defined for matrices of same size
- **Commutative**: $A + B = B + A$
- **Associative**: $(A + B) + C = A + (B + C)$

Scalar Multiplication: $(cA)_{ij} = c \cdot a_{ij}$

- **Distributive**: $c(A + B) = cA + cB$
- $(c + d)A = cA + dA$

Matrix Multiplication: $(AB)_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$

For $A_{m \times p}$ and $B_{p \times n}$, result is $C_{m \times n}$

Properties:

- **Not commutative**: Generally $AB \neq BA$
- **Associative**: $(AB)C = A(BC)$
- **Distributive**: $A(B + C) = AB + AC$
- **Identity**: $AI = IA = A$

Transpose: $(A^T)_{ij} = a_{ji}$

Properties of Transpose:

- $(A^T)^T = A$
- $(A + B)^T = A^T + B^T$
- $(AB)^T = B^T A^T$ (order reverses!)
- $(cA)^T = cA^T$

Determinants:

2×2 Determinant:

$$\det(A) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

3×3 Determinant (Cofactor expansion along first row):

$$\det(A) = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

General Cofactor Expansion:

$$\det(A) = \sum_{j=1}^{n} a_{ij} C_{ij} = \sum_{i=1}^{n} a_{ij} C_{ij}$$

where $C_{ij} = (-1)^{i+j} M_{ij}$ is the cofactor and M_{ij} is the minor

Properties of Determinants:

1. **Row/Column Operations**:
   - Swapping two rows/columns changes sign
   - Multiplying row/column by scalar k multiplies determinant by k
   - Adding multiple of one row to another doesn't change determinant
2. **Special Cases**:
   - $\det(I) = 1$
   - $\det(A^T) = \det(A)$
   - $\det(AB) = \det(A)\det(B)$
   - $\det(cA) = c^n \det(A)$ for $n \times n$ matrix
3. **Triangular Matrices**: Determinant equals product of diagonal entries
4. **Zero Determinant**: Matrix is singular (non-invertible) iff $\det(A) = 0$

Cramer's Rule:

For system $Ax = b$ where A is $n \times n$ and $\det(A) \neq 0$:

$$x_i = \frac{\det(A_i)}{\det(A)}$$

where A_i is matrix A with i-th column replaced by b

Matrix Inverse:

Definition: A^{-1} is inverse of A if $AA^{-1} = A^{-1}A = I$

Existence: A^{-1} exists iff $\det(A) \neq 0$ (A is non-singular)

2×2 Inverse:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \text{ for } A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

General Formula (Adjugate method):

$$A^{-1} = \frac{1}{\det(A)} \text{adj}(A)$$

where $\text{adj}(A) = [C_{ji}]$ (transpose of cofactor matrix)

Properties of Inverse:

- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$ (order reverses!)
- $(A^T)^{-1} = (A^{-1})^T$
- $\det(A^{-1}) = \frac{1}{\det(A)}$

Elementary Matrices:

Matrices representing elementary row operations:

1. **Row swap**: E_{ij} (swaps rows i and j)
2. **Row scaling**: $E_i(c)$ (multiplies row i by $c \neq 0$)
3. **Row addition**: $E_{ij}(c)$ (adds c times row j to row i)

Theorem: Every invertible matrix is product of elementary matrices

Gauss-Jordan Elimination:

Method to find A^{-1} by row operations:

1. Form augmented matrix $[A|I]$
2. Use row operations to transform to $[I|A^{-1}]$
3. If process fails, A is not invertible

Rank of Matrix:

Definition: Rank is maximum number of linearly independent rows (or columns)

Properties:

- $\text{rank}(A) = \text{rank}(A^T)$
- $\text{rank}(A) \leq \min(m, n)$ for $m \times n$ matrix
- A is invertible iff $\text{rank}(A) = n$ (full rank)

Computing Rank:

1. Use row operations to get row echelon form
2. Count non-zero rows

Applications:

System of Linear Equations: $Ax = b$

- **Unique solution**: $\det(A) \neq 0$ (use Cramer's rule or $x = A^{-1}b$)
- **No solution or infinitely many**: $\det(A) = 0$

Consistency Conditions:

- **Consistent**: $\text{rank}(A) = \text{rank}([A|b])$
- **Unique solution**: $\text{rank}(A) = n$ (number of variables)
- **Infinitely many solutions**: $\text{rank}(A) < n$

Geometric Interpretations:

- **Determinant**: Signed volume of parallelepiped formed by column vectors
- **Matrix multiplication**: Composition of linear transformations
- **Inverse**: Reverse transformation

Problem-Solving Tips:

For Determinants:

1. Use row operations to simplify before expanding
2. Look for zeros to minimize calculation
3. For large matrices, use LU decomposition

For Matrix Inverse:

1. Check if $\det(A) \neq 0$ first
2. For 2×2, use direct formula
3. For larger matrices, use Gauss-Jordan elimination

For Systems:

1. Check consistency using rank
2. If unique solution exists, use appropriate method
3. For homogeneous systems, non-trivial solution exists iff $\det(A) = 0$

GATE Tips:

- $(AB)^T = B^T A^T$ and $(AB)^{-1} = B^{-1} A^{-1}$ (order reverses!)
- $\det(AB) = \det(A)\det(B)$
- Row operations: swap changes sign, scaling multiplies by scalar
- Cramer's rule only when $\det(A) \neq 0$
- Rank determines solution type for linear systems
- Elementary matrices represent row operations

Examples:

1. **Find determinant** of $A = \begin{bmatrix} 2 & 1 & 3 \\ 0 & 4 & 1 \\ 0 & 0 & 5 \end{bmatrix}$:
   - Upper triangular matrix
   - $\det(A) = 2 \times 4 \times 5 = 40$
2. **Find inverse** of $A = \begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}$:
   - $\det(A) = 2(2) - 1(3) = 1$
   - $A^{-1} = \frac{1}{1}\begin{bmatrix} 2 & -1 \\ -3 & 2 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -3 & 2 \end{bmatrix}$
3. **Solve system** using Cramer's rule: $2x + y = 5$, $x + 3y = 8$:
   - $A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$, $b = \begin{bmatrix} 5 \\ 8 \end{bmatrix}$
   - $\det(A) = 6 - 1 = 5$

- $x = \frac{\begin{vmatrix} 5 & 1 \\ 8 & 3 \end{vmatrix}}{5} = \frac{15-8}{5} = \frac{7}{5}$
- $y = \frac{\begin{vmatrix} 2 & 5 \\ 1 & 8 \end{vmatrix}}{5} = \frac{16-5}{5} = \frac{11}{5}$

6.2 Vector Spaces (25)

Key Concepts: Vector spaces are algebraic structures where vectors can be added and scaled. They provide the foundation for linear algebra and functional analysis.

Vector Space Definition:

A **vector space** V over field F (usually $\mathbb{R}$ or $\mathbb{C}$) is set with two operations:

- **Vector addition**: $+ : V \times V \to V$
- **Scalar multiplication**: $\cdot : F \times V \to V$

satisfying these axioms for all $\mathbf{u}, \mathbf{v}, \mathbf{w} \in V$ and $a, b \in F$:

Addition Axioms:

1. **Closure**: $\mathbf{u} + \mathbf{v} \in V$
2. **Commutativity**: $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$
3. **Associativity**: $(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$
4. **Zero vector**: $\exists \mathbf{0} \in V$ such that $\mathbf{v} + \mathbf{0} = \mathbf{v}$
5. **Additive inverse**: $\forall \mathbf{v} \in V, \exists (-\mathbf{v}) \in V$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$

Scalar Multiplication Axioms:
6. **Closure**: $a\mathbf{v} \in V$
7. **Associativity**: $a(b\mathbf{v}) = (ab)\mathbf{v}$
8. **Identity**: $1\mathbf{v} = \mathbf{v}$
9. **Distributivity**: $a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$
10. **Distributivity**: $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$

Examples of Vector Spaces:

1. $\mathbb{R}^n$: n-tuples of real numbers with componentwise operations
2. $\mathbb{C}^n$: n-tuples of complex numbers
3. $M_{m \times n}(\mathbb{R})$: $m \times n$ real matrices
4. $P_n(\mathbb{R})$: Polynomials of degree ≤ n with real coefficients
5. $C[a, b]$: Continuous functions on interval $[a, b]$
6. $\{0\}$: Trivial vector space containing only zero vector

Subspaces:

Definition: Subset $W \subseteq V$ is **subspace** if:

1. $\mathbf{0} \in W$ (contains zero vector)
2. **Closed under addition**: $\mathbf{u}, \mathbf{v} \in W \Rightarrow \mathbf{u} + \mathbf{v} \in W$
3. **Closed under scalar multiplication**: $\mathbf{v} \in W, a \in F \Rightarrow a\mathbf{v} \in W$

Subspace Test: W is subspace iff $a\mathbf{u} + b\mathbf{v} \in W$ for all $\mathbf{u}, \mathbf{v} \in W$ and $a, b \in F$

Examples of Subspaces:

- Lines through origin in $\mathbb{R}^2$
- Planes through origin in $\mathbb{R}^3$
- Solution space of homogeneous system $A\mathbf{x} = \mathbf{0}$
- Even polynomials in $P_n(\mathbb{R})$

Linear Combinations:

Definition: Vector $\mathbf{v}$ is **linear combination** of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ if:

$$\mathbf{v} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_k\mathbf{v}_k$$

for some scalars $a_1, \ldots, a_k$

Span: $\text{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ is set of all linear combinations of $\mathbf{v}_1, \ldots, \mathbf{v}_k$

Properties:

- $\text{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ is always a subspace
- Smallest subspace containing $\mathbf{v}_1, \ldots, \mathbf{v}_k$

Linear Independence:

Definition: Vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are **linearly independent** if:

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_k\mathbf{v}_k = \mathbf{0} \Rightarrow a_1 = a_2 = \cdots = a_k = 0$$

Linear Dependence: Not linearly independent; some non-trivial linear combination equals zero

Tests for Linear Independence:

1. **Matrix method**: Form matrix with vectors as columns; independent iff columns are linearly independent
2. **Determinant**: For square matrix, independent iff $\det \neq 0$
3. **Row reduction**: Independent iff no free variables in reduced form

Properties:

- Any set containing zero vector is linearly dependent
- Two vectors are dependent iff one is scalar multiple of other
- In $\mathbb{R}^n$, at most n vectors can be linearly independent

Basis and Dimension:

Basis: Set of vectors $\mathcal{B} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ is **basis** for V if:

1. $\mathcal{B}$ is linearly independent
2. $\text{span}(\mathcal{B}) = V$

Properties of Bases:

- Every vector in V has unique representation as linear combination of basis vectors
- All bases of finite-dimensional vector space have same number of elements

Dimension: $\dim(V)$ is number of vectors in any basis of V

Standard Bases:

- $\mathbb{R}^n$: $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$ where $\mathbf{e}_i$ has 1 in i-th position, 0 elsewhere
- $P_n(\mathbb{R})$: $\{1, x, x^2, \ldots, x^n\}$
- $M_{2\times 2}(\mathbb{R})$: $\left\{\begin{bmatrix}1 & 0\\0 & 0\end{bmatrix}, \begin{bmatrix}0 & 1\\0 & 0\end{bmatrix}, \begin{bmatrix}0 & 0\\1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\right\}$

Fundamental Theorems:

Dimension Theorem: If V is finite-dimensional with $\dim(V) = n$, then:

- Any linearly independent set has at most n vectors
- Any spanning set has at least n vectors
- Any linearly independent set of n vectors is a basis
- Any spanning set of n vectors is a basis

Rank-Nullity Theorem: For linear transformation $T : V \to W$:

$$\dim(V) = \dim(\ker(T)) + \dim(\text{range}(T))$$

Coordinate Systems:

Given basis $\mathcal{B} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ for V, every $\mathbf{v} \in V$ has unique representation:

$$\mathbf{v} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n$$

Coordinate vector: $[\mathbf{v}]_{\mathcal{B}} = \begin{bmatrix}c_1\\c_2\\\vdots\\c_n\end{bmatrix}$

Change of Basis:

For bases $\mathcal{B}$ and $\mathcal{C}$, **change of basis matrix** $P_{\mathcal{C}\leftarrow\mathcal{B}}$ satisfies:

$$[\mathbf{v}]_{\mathcal{C}} = P_{\mathcal{C}\leftarrow\mathcal{B}}[\mathbf{v}]_{\mathcal{B}}$$

Construction: Columns of $P_{\mathcal{C}\leftarrow\mathcal{B}}$ are coordinate vectors of $\mathcal{B}$-basis vectors with respect to $\mathcal{C}$

Row Space and Column Space:

For matrix A:

- **Row space**: $\text{row}(A) = \text{span of row vectors}$
- **Column space**: $\text{col}(A) = \text{span of column vectors}$
- **Null space**: $\text{null}(A) = \{\mathbf{x} : A\mathbf{x} = \mathbf{0}\}$

Properties:

- $\text{rank}(A) = \dim(\text{row}(A)) = \dim(\text{col}(A))$
- $\text{nullity}(A) = \dim(\text{null}(A))$
- $\text{rank}(A) + \text{nullity}(A) = n$ (number of columns)

Inner Product Spaces:

Inner Product: Function $\langle \cdot, \cdot \rangle : V \times V \to F$ satisfying:

1. **Linearity**: $\langle a\mathbf{u} + b\mathbf{v}, \mathbf{w} \rangle = a\langle \mathbf{u}, \mathbf{w} \rangle + b\langle \mathbf{v}, \mathbf{w} \rangle$
2. **Conjugate symmetry**: $\langle \mathbf{u}, \mathbf{v} \rangle = \overline{\langle \mathbf{v}, \mathbf{u} \rangle}$
3. **Positive definiteness**: $\langle \mathbf{v}, \mathbf{v} \rangle \geq 0$ with equality iff $\mathbf{v} = \mathbf{0}$

Standard Inner Products:

- $\mathbb{R}^n$: $\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i$
- $\mathbb{C}^n$: $\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{n} u_i \overline{v_i}$
- $C[a, b]$: $\langle f, g \rangle = \int_a^b f(x)g(x)dx$

Norm: $\|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}$

Orthogonality: Vectors $\mathbf{u}, \mathbf{v}$ are **orthogonal** if $\langle \mathbf{u}, \mathbf{v} \rangle = 0$

Orthogonal and Orthonormal Sets:

Orthogonal set: Pairwise orthogonal vectors
Orthonormal set: Orthogonal set where each vector has norm 1

Properties:

- Orthogonal sets are linearly independent (except for zero vector)
- Orthonormal basis simplifies calculations

Gram-Schmidt Process:

Algorithm to convert basis $\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ to orthonormal basis $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$:

1. $\mathbf{u}_1 = \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|}$

2. For $k = 2, 3, \ldots, n$:
   - $\mathbf{w}_k = \mathbf{v}_k - \sum_{j=1}^{k-1} \langle \mathbf{v}_k, \mathbf{u}_j \rangle \mathbf{u}_j$
   - $\mathbf{u}_k = \frac{\mathbf{w}_k}{\|\mathbf{w}_k\|}$

Problem-Solving Tips:

For Subspaces:

1. Check if zero vector is included
2. Verify closure under addition and scalar multiplication
3. Use subspace test: $a\mathbf{u} + b\mathbf{v} \in W$

For Linear Independence:

1. Set up equation $a_1\mathbf{v}_1 + \cdots + a_k\mathbf{v}_k = \mathbf{0}$
2. Solve for coefficients
3. Independent iff only trivial solution

For Basis:

1. Check linear independence
2. Check if span equals the space
3. Count vectors (should equal dimension)

GATE Tips:

- Subspace must contain zero vector and be closed under operations
- Linear independence: only trivial combination gives zero
- Basis: linearly independent spanning set
- Dimension: number of vectors in basis
- Rank-nullity: $\text{rank} + \text{nullity} = \text{number of columns}$
- Orthogonal vectors are linearly independent
- Gram-Schmidt produces orthonormal basis

Examples:

1. **Check if subspace**: $W = \{(x, y, z) \in \mathbb{R}^3 : x + y + z = 0\}$
   - Contains $(0, 0, 0)$: $0 + 0 + 0 = 0$ ✓
   - Closed under addition: If $x_1 + y_1 + z_1 = 0$ and $x_2 + y_2 + z_2 = 0$, then $(x_1 + x_2) + (y_1 + y_2) + (z_1 + z_2) = 0$ ✓
   - Closed under scalar multiplication: If $x + y + z = 0$, then $cx + cy + cz = c(x + y + z) = 0$ ✓
   - Therefore W is subspace
2. **Test linear independence**: $\mathbf{v}_1 = (1, 2, 1)$, $\mathbf{v}_2 = (2, 1, 3)$, $\mathbf{v}_3 = (1, -1, 2)$
   - Set up: $a(1, 2, 1) + b(2, 1, 3) + c(1, -1, 2) = (0, 0, 0)$

- System: $a + 2b + c = 0$, $2a + b - c = 0$, $a + 3b + 2c = 0$
- Solving: $a = b = c = 0$ (only solution)
- Therefore vectors are linearly independent

3. **Find basis for null space** of $A = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 2 & 4 & 1 & 1 \end{bmatrix}$:
   - Row reduce: $\begin{bmatrix} 1 & 2 & 0 & -1 \\ 0 & 0 & 1 & 2 \end{bmatrix}$
   - Free variables: x_2, x_4
   - General solution: $x_1 = -2x_2 + x_4$, $x_3 = -2x_4$
   - Basis: $\{(-2, 1, 0, 0), (1, 0, -2, 1)\}$

6.3 Eigenvalues and Eigenvectors (30)

Key Concepts: Eigenvalues and eigenvectors reveal the fundamental directions and scaling factors of linear transformations. They are crucial for understanding matrix behavior and solving differential equations.

Definitions:

Eigenvector: Non-zero vector $\mathbf{v}$ such that $A\mathbf{v} = \lambda\mathbf{v}$ for some scalar λ

Eigenvalue: Scalar λ such that $A\mathbf{v} = \lambda\mathbf{v}$ for some non-zero vector $\mathbf{v}$

Geometric Interpretation: Eigenvectors are directions that remain unchanged (only scaled) under the linear transformation represented by A

Characteristic Equation:

From $A\mathbf{v} = \lambda\mathbf{v}$:

$$(A - \lambda I)\mathbf{v} = \mathbf{0}$$

For non-trivial solution, $(A - \lambda I)$ must be singular:

$$\det(A - \lambda I) = 0$$

This is the **characteristic equation** or **characteristic polynomial**

Finding Eigenvalues and Eigenvectors:

Step 1: Find eigenvalues by solving $\det(A - \lambda I) = 0$

Step 2: For each eigenvalue λ_i, find eigenvectors by solving $(A - \lambda_i I)\mathbf{v} = \mathbf{0}$

Example: Find eigenvalues and eigenvectors of $A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$

Step 1: Characteristic equation

$$\det(A - \lambda I) = \det \begin{bmatrix} 3-\lambda & 1 \\ 0 & 2-\lambda \end{bmatrix} = (3-\lambda)(2-\lambda) = 0$$

Eigenvalues: $\lambda_1 = 3, \lambda_2 = 2$

Step 2: Find eigenvectors

For $\lambda_1 = 3$:

$$(A - 3I)\mathbf{v} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This gives $v_2 = 0$, so eigenvector: $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ (or any scalar multiple)

For $\lambda_2 = 2$:

$$(A - 2I)\mathbf{v} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This gives $v_1 + v_2 = 0$, so eigenvector: $\mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

Properties of Eigenvalues:

1. **Sum of eigenvalues** = trace of matrix: $\sum \lambda_i = \text{tr}(A) = \sum a_{ii}$
2. **Product of eigenvalues** = determinant: $\prod \lambda_i = \det(A)$
3. **Eigenvalues of special matrices**:
   - Triangular matrix: eigenvalues are diagonal entries
   - Symmetric matrix: all eigenvalues are real
   - Orthogonal matrix: $|\lambda_i| = 1$
4. **Powers**: If λ is eigenvalue of A, then λ^k is eigenvalue of A^k
5. **Inverse**: If A is invertible and $\lambda \neq 0$ is eigenvalue of A, then $1/\lambda$ is eigenvalue of A^{-1}

Algebraic and Geometric Multiplicity:

Algebraic Multiplicity: Multiplicity of eigenvalue as root of characteristic polynomial

Geometric Multiplicity: Dimension of eigenspace (null space of $A - \lambda I$)

Relationship: $1 \leq \text{geometric multiplicity} \leq \text{algebraic multiplicity}$

Eigenspaces:

Eigenspace E_λ: Set of all eigenvectors corresponding to eigenvalue λ, plus zero vector

$$E_\lambda = \text{null}(A - \lambda I) = \{\mathbf{v} : (A - \lambda I)\mathbf{v} = \mathbf{0}\}$$

Properties:

- E_λ is subspace

- $\dim(E_\lambda)$ = geometric multiplicity of λ
- Eigenvectors from different eigenspaces are linearly independent

Diagonalization:

Definition: Matrix A is **diagonalizable** if there exists invertible matrix P such that:

$$P^{-1}AP = D$$

where D is diagonal matrix

Diagonalization Theorem: $n \times n$ matrix A is diagonalizable iff:

- A has n linearly independent eigenvectors
- For each eigenvalue, geometric multiplicity = algebraic multiplicity

Diagonalization Process:

1. Find all eigenvalues and eigenvectors
2. Check if there are n linearly independent eigenvectors
3. Form matrix P with eigenvectors as columns
4. $D = P^{-1}AP$ has eigenvalues on diagonal

Benefits of Diagonalization:

- **Powers**: $A^k = PD^kP^{-1}$ where D^k is easy to compute
- **Matrix functions**: $f(A) = Pf(D)P^{-1}$
- **Systems of differential equations**: Decouples the system

Symmetric Matrices:

Spectral Theorem: Every real symmetric matrix is orthogonally diagonalizable

$$A = QDQ^T$$

where Q is orthogonal ($Q^TQ = I$) and D is diagonal

Properties:

- All eigenvalues are real
- Eigenvectors from different eigenvalues are orthogonal
- Always diagonalizable

Orthogonal Diagonalization Process:

1. Find eigenvalues and eigenvectors
2. Use Gram-Schmidt to orthonormalize eigenvectors within each eigenspace
3. Form orthogonal matrix Q with orthonormal eigenvectors as columns

Quadratic Forms:

Quadratic form: $Q(\mathbf{x}) = \mathbf{x}^T A\mathbf{x}$ where A is symmetric

Classification (based on eigenvalues of A):

- **Positive definite**: All eigenvalues > 0
- **Positive semidefinite**: All eigenvalues ≥ 0
- **Negative definite**: All eigenvalues < 0
- **Negative semidefinite**: All eigenvalues ≤ 0
- **Indefinite**: Both positive and negative eigenvalues

Principal Component Analysis (PCA):

Statistical technique using eigenvalue decomposition:

1. Center data by subtracting mean
2. Compute covariance matrix C
3. Find eigenvalues and eigenvectors of C
4. Principal components are eigenvectors with largest eigenvalues

Applications:

1. Systems of Differential Equations:
For $\mathbf{x}' = A\mathbf{x}$, if A is diagonalizable:

- Solution: $\mathbf{x}(t) = c_1 e^{\lambda_1 t}\mathbf{v}_1 + \cdots + c_n e^{\lambda_n t}\mathbf{v}_n$

2. Markov Chains:

- Steady state corresponds to eigenvector with eigenvalue 1
- Convergence rate determined by second-largest eigenvalue

3. Vibration Analysis:

- Natural frequencies are square roots of eigenvalues
- Mode shapes are eigenvectors

4. Google PageRank:

- Web page importance from dominant eigenvector of link matrix

Computational Methods:

Power Method: Iterative method for finding dominant eigenvalue

1. Start with initial vector $\mathbf{x}_0$
2. Iterate: $\mathbf{x}_{k+1} = A\mathbf{x}_k$ (with normalization)

3. Converges to eigenvector of largest eigenvalue

QR Algorithm: Advanced method for finding all eigenvalues

- Repeatedly applies QR decomposition
- Converges to upper triangular form with eigenvalues on diagonal

Problem-Solving Tips:

For Finding Eigenvalues:

1. Set up characteristic equation $\det(A - \lambda I) = 0$
2. Expand determinant carefully
3. Solve polynomial equation

For Finding Eigenvectors:

1. Substitute each eigenvalue into $(A - \lambda I)\mathbf{v} = \mathbf{0}$
2. Solve homogeneous system
3. Express solution in terms of free variables

For Diagonalization:

1. Check if n linearly independent eigenvectors exist
2. Form matrix P with eigenvectors as columns
3. Verify $AP = PD$

GATE Tips:

- Sum of eigenvalues = trace, product = determinant
- Symmetric matrices have real eigenvalues and orthogonal eigenvectors
- Geometric multiplicity ≤ algebraic multiplicity
- Matrix is diagonalizable iff geometric = algebraic multiplicity for all eigenvalues
- Eigenvectors from different eigenvalues are linearly independent
- For triangular matrices, eigenvalues are diagonal entries

Examples:

1. **Find eigenvalues** of $A = \begin{bmatrix} 4 & -2 \\ 1 & 1 \end{bmatrix}$:
   - $\det(A - \lambda I) = \det \begin{bmatrix} 4-\lambda & -2 \\ 1 & 1-\lambda \end{bmatrix} = (4-\lambda)(1-\lambda) + 2 = \lambda^2 - 5\lambda + 6 = (\lambda - 2)(\lambda - 3)$
   - Eigenvalues: $\lambda_1 = 2, \lambda_2 = 3$
2. **Check diagonalizability** of $A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$:
   - Characteristic polynomial: $(1 - \lambda)^3$

- Only eigenvalue: $\lambda = 1$ (algebraic multiplicity 3)
- $(A - I) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$
- Null space has dimension 1 (geometric multiplicity 1)
- Since $1 < 3$, matrix is not diagonalizable

3. **Orthogonally diagonalize** $A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$:
   - Eigenvalues: $\lambda_1 = 4, \lambda_2 = 2$
   - Eigenvectors: $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$
   - Normalize: $\mathbf{u}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{u}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$
   - $Q = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, D = \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix}$

6.4 Linear Transformations (25)

Key Concepts: Linear transformations are functions between vector spaces that preserve vector addition and scalar multiplication. They provide the connection between abstract linear algebra and concrete matrix operations.

Definition:

Linear Transformation: Function $T : V \to W$ between vector spaces satisfying:

1. **Additivity**: $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$
2. **Homogeneity**: $T(c\mathbf{v}) = cT(\mathbf{v})$

Equivalent condition: $T(c\mathbf{u} + d\mathbf{v}) = cT(\mathbf{u}) + dT(\mathbf{v})$

Properties:

- $T(\mathbf{0}) = \mathbf{0}$ (zero vector maps to zero vector)
- $T(-\mathbf{v}) = -T(\mathbf{v})$
- T preserves linear combinations

Matrix Representation:

For finite-dimensional spaces with bases $\mathcal{B} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ for V and $\mathcal{C} = \{\mathbf{w}_1, \ldots, \mathbf{w}_m\}$ for W :

Matrix of transformation: $[T]_{\mathcal{C}}^{\mathcal{B}}$ is $m \times n$ matrix where column j is $[T(\mathbf{v}_j)]_{\mathcal{C}}$

Coordinate relationship: $[T(\mathbf{v})]_{\mathcal{C}} = [T]_{\mathcal{C}}^{\mathcal{B}} [\mathbf{v}]_{\mathcal{B}}$

Standard Matrix: For $T : \mathbb{R}^n \to \mathbb{R}^m$ with standard bases:

$$[T] = [T(\mathbf{e}_1)|T(\mathbf{e}_2)|\cdots|T(\mathbf{e}_n)]$$

Examples of Linear Transformations:

1. Geometric Transformations in $\mathbb{R}^2$:

Rotation by angle θ:

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Reflection across x-axis:

$$\text{Ref}_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Scaling by factors s_x, s_y:

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Shear parallel to x-axis:

$$\text{Shear}_x = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

Projection onto x-axis:

$$P_x = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

2. Differentiation: $D : P_n(\mathbb{R}) \to P_{n-1}(\mathbb{R})$ where $D(p(x)) = p'(x)$

3. Integration: $I : C[a,b] \to \mathbb{R}$ where $I(f) = \int_a^b f(x)dx$

4. Matrix Multiplication: $T_A : \mathbb{R}^n \to \mathbb{R}^m$ where $T_A(\mathbf{x}) = A\mathbf{x}$

Kernel and Range:

Kernel (Null Space): $\ker(T) = \{\mathbf{v} \in V : T(\mathbf{v}) = \mathbf{0}\}$

- Always a subspace of V
- T is injective iff $\ker(T) = \{\mathbf{0}\}$

Range (Image): $\text{range}(T) = \{T(\mathbf{v}) : \mathbf{v} \in V\}$

- Always a subspace of W
- T is surjective iff $\text{range}(T) = W$

Rank-Nullity Theorem:

$$\dim(V) = \dim(\ker(T)) + \dim(\text{range}(T))$$

$$\text{nullity}(T) + \text{rank}(T) = \dim(V)$$

Proof Idea: Choose basis for $\ker(T)$, extend to basis for V, show images of extension vectors form basis for $\text{range}(T)$

Isomorphisms:

Isomorphism: Bijective linear transformation

- T is isomorphism iff T is both injective and surjective
- Equivalent: $\ker(T) = \{\mathbf{0}\}$ and $\text{range}(T) = W$

Properties:

- If $T : V \to W$ is isomorphism, then $\dim(V) = \dim(W)$
- Inverse $T^{-1} : W \to V$ is also linear transformation
- Composition of isomorphisms is isomorphism

Fundamental Theorem: Two finite-dimensional vector spaces over same field are isomorphic iff they have same dimension

Change of Basis for Linear Transformations:

If $T : V \to V$ with bases $\mathcal{B}$ and $\mathcal{C}$:

$$[T]_{\mathcal{C}} = P^{-1}[T]_{\mathcal{B}}P$$

where P is change of basis matrix from $\mathcal{B}$ to $\mathcal{C}$

Similar Matrices: Matrices A and B are **similar** if $B = P^{-1}AP$ for some invertible P

Properties of Similar Matrices:

- Same eigenvalues (with same multiplicities)
- Same determinant and trace
- Same rank
- Represent same linear transformation in different bases

Composition of Linear Transformations:

For $T : U \to V$ and $S : V \to W$:
Composition: $(S \circ T) : U \to W$ where $(S \circ T)(\mathbf{u}) = S(T(\mathbf{u}))$

Matrix representation: $[S \circ T] = [S][T]$

Properties:

- Composition is associative: $(R \circ S) \circ T = R \circ (S \circ T)$
- Generally not commutative: $S \circ T \neq T \circ S$

Inverse Transformations:

If $T : V \to W$ is isomorphism, then $T^{-1} : W \to V$ exists and:

- T^{-1} is linear
- $T \circ T^{-1} = I_W$ and $T^{-1} \circ T = I_V$
- $[T^{-1}] = [T]^{-1}$

Orthogonal Transformations:

Definition: Linear transformation $T : \mathbb{R}^n \to \mathbb{R}^n$ is **orthogonal** if it preserves inner products:

$$\langle T(\mathbf{u}), T(\mathbf{v}) \rangle = \langle \mathbf{u}, \mathbf{v} \rangle$$

Equivalent conditions:

- T preserves lengths: $\|T(\mathbf{v})\| = \|\mathbf{v}\|$
- Matrix representation is orthogonal: $A^T A = I$
- Columns (and rows) form orthonormal set

Examples: Rotations, reflections

Projections:

Orthogonal Projection onto subspace W:
For $\mathbf{v} \in V$, $\text{proj}_W(\mathbf{v})$ is closest point in W to $\mathbf{v}$

Formula: If $\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$ is orthonormal basis for W:

$$\text{proj}_W(\mathbf{v}) = \sum_{i=1}^{k} \langle \mathbf{v}, \mathbf{u}_i \rangle \mathbf{u}_i$$

Matrix form: $P = UU^T$ where U has orthonormal basis vectors as columns

Properties:

- $P^2 = P$ (idempotent)
- $P^T = P$ (symmetric)
- $\text{range}(P) = W$, $\ker(P) = W^\perp$

Least Squares Solutions:

For inconsistent system $A\mathbf{x} = \mathbf{b}$, **least squares solution** minimizes $\|A\mathbf{x} - \mathbf{b}\|^2$

Normal equation: $A^T A\mathbf{x} = A^T \mathbf{b}$

Solution: $\mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}$ (if $A^T A$ is invertible)

Geometric interpretation: Project $\mathbf{b}$ onto column space of A

Applications:

1. Computer Graphics:

- Transformations for 2D/3D graphics
- Homogeneous coordinates for translations
- Perspective projections

2. Data Analysis:

- Principal Component Analysis (PCA)
- Linear regression
- Dimensionality reduction

3. Differential Equations:

- Systems of linear ODEs
- Stability analysis using eigenvalues

4. Signal Processing:

- Fourier transforms
- Filter design
- Image processing

5. Quantum Mechanics:

- State transformations
- Measurement operators
- Unitary evolution

Problem-Solving Tips:

To verify linearity:

1. Check $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$
2. Check $T(c\mathbf{v}) = cT(\mathbf{v})$
3. Or verify combined condition

To find matrix representation:

1. Apply T to each basis vector
2. Express results in terms of target basis
3. Use coefficients as matrix columns

To find kernel and range:

1. Kernel: Solve $T(\mathbf{v}) = \mathbf{0}$ or $A\mathbf{x} = \mathbf{0}$

2. Range: Find span of T applied to basis vectors

GATE Tips:

- Linear transformation preserves linear combinations
- Matrix columns are images of standard basis vectors
- Rank-nullity: $\dim(V) = \text{nullity} + \text{rank}$
- Isomorphism iff bijective iff $\ker(T) = \{\mathbf{0}\}$ and onto
- Similar matrices have same eigenvalues
- Orthogonal transformations preserve lengths and angles
- Projection matrices are idempotent and symmetric

Examples:

1. **Find matrix** of rotation by 90° in $\mathbb{R}^2$:
   - $T(\mathbf{e}_1) = T\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$
   - $T(\mathbf{e}_2) = T\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}-1\\0\end{bmatrix}$
   - Matrix: $\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}$
2. **Find kernel** of $T : \mathbb{R}^3 \to \mathbb{R}^2$ where $T\begin{bmatrix}x\\y\\z\end{bmatrix} = \begin{bmatrix}x+y\\y+z\end{bmatrix}$:
   - Matrix: $A = \begin{bmatrix}1 & 1 & 0\\0 & 1 & 1\end{bmatrix}$
   - Solve $A\mathbf{x} = \mathbf{0}$: $x + y = 0$, $y + z = 0$
   - Solution: $x = -y$, $z = -y$
   - Kernel: $\text{span}\left\{\begin{bmatrix}-1\\1\\-1\end{bmatrix}\right\}$
3. **Verify rank-nullity** for above transformation:
   - $\dim(\mathbb{R}^3) = 3$
   - $\text{nullity}(T) = 1$ (dimension of kernel)
   - $\text{rank}(T) = 2$ (dimension of range)
   - Check: $3 = 1 + 2$ ✓

7. Engineering Mathematics: Probability and Statistics (100 Questions)

7.1 Probability Theory (40)

Key Concepts: Probability theory provides mathematical framework for analyzing uncertainty and randomness. It forms the foundation for statistics, machine learning, and decision theory.

Sample Space and Events:

Sample Space Ω: Set of all possible outcomes of an experiment

- **Discrete**: Finite or countably infinite (coin flips, dice rolls)
- **Continuous**: Uncountably infinite (measurements, time intervals)

Event: Subset of sample space

- **Elementary event**: Single outcome
- **Compound event**: Union of elementary events
- **Certain event**: Ω (always occurs)
- **Impossible event**: $\emptyset$ (never occurs)

Event Operations:

- **Union**: $A \cup B$ (A or B occurs)
- **Intersection**: $A \cap B$ (both A and B occur)
- **Complement**: A^c (A does not occur)
- **Difference**: $A - B = A \cap B^c$

De Morgan's Laws:

- $(A \cup B)^c = A^c \cap B^c$
- $(A \cap B)^c = A^c \cup B^c$

Probability Axioms (Kolmogorov):

For probability function $P : \mathcal{F} \to [0, 1]$ where $\mathcal{F}$ is σ-algebra of events:

1. **Non-negativity**: $P(A) \geq 0$ for all events A
2. **Normalization**: $P(\Omega) = 1$
3. **Countable additivity**: For disjoint events $A_1, A_2, \ldots$:

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$$

Basic Properties:

- $P(\emptyset) = 0$
- $P(A^c) = 1 - P(A)$
- If $A \subseteq B$, then $P(A) \leq P(B)$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

Inclusion-Exclusion Principle:

$$P(A_1 \cup A_2 \cup A_3) = \sum P(A_i) - \sum P(A_i \cap A_j) + P(A_1 \cap A_2 \cap A_3)$$

Classical Probability:

For finite sample space with equally likely outcomes:

$$P(A) = \frac{|A|}{|\Omega|} = \frac{\text{number of favorable outcomes}}{\text{total number of outcomes}}$$

Examples:

- Fair coin: $P(\text{Heads}) = \frac{1}{2}$
- Fair die: $P(\text{even number}) = \frac{3}{6} = \frac{1}{2}$
- Card from deck: $P(\text{Ace}) = \frac{4}{52} = \frac{1}{13}$

Conditional Probability:

Definition: Probability of event A given that event B has occurred:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \quad \text{provided } P(B) > 0$$

Interpretation: Restricts sample space to outcomes where B occurs

Properties:

- $P(A|B) \geq 0$
- $P(\Omega|B) = 1$
- $P(A_1 \cup A_2|B) = P(A_1|B) + P(A_2|B) - P(A_1 \cap A_2|B)$

Multiplication Rule:

$$P(A \cap B) = P(A|B) \cdot P(B) = P(B|A) \cdot P(A)$$

Chain Rule: For events $A_1, A_2, \ldots, A_n$:

$$P(A_1 \cap A_2 \cap \cdots \cap A_n) = P(A_1) \cdot P(A_2|A_1) \cdot P(A_3|A_1 \cap A_2) \cdots P(A_n|A_1 \cap \cdots \cap A_{n-1})$$

Independence:

Definition: Events A and B are **independent** if:

$$P(A \cap B) = P(A) \cdot P(B)$$

Equivalent conditions (when $P(B) > 0$):

- $P(A|B) = P(A)$
- $P(B|A) = P(B)$

Mutual Independence: Events $A_1, \ldots, A_n$ are mutually independent if for every subset $\{i_1, \ldots, i_k\}$:

$$P(A_{i_1} \cap \cdots \cap A_{i_k}) = P(A_{i_1}) \cdots P(A_{i_k})$$

Pairwise vs Mutual Independence: Pairwise independence doesn't imply mutual independence

Law of Total Probability:

If $\{B_1, B_2, \ldots, B_n\}$ is partition of Ω (disjoint and exhaustive), then:

$$P(A) = \sum_{i=1}^{n} P(A|B_i) \cdot P(B_i)$$

Continuous version: $P(A) = \int P(A|B_x) \cdot f_B(x)dx$

Bayes' Theorem:

Statement: For partition $\{B_1, \ldots, B_n\}$ and event A with $P(A) > 0$:

$$P(B_j|A) = \frac{P(A|B_j) \cdot P(B_j)}{\sum_{i=1}^{n} P(A|B_i) \cdot P(B_i)}$$

Interpretation:

- $P(B_j)$: **Prior probability** (before observing A)
- $P(A|B_j)$: **Likelihood** (probability of evidence given hypothesis)
- $P(B_j|A)$: **Posterior probability** (after observing A)

Applications:

- Medical diagnosis
- Spam filtering
- Machine learning classification

Random Variables:

Definition: Function $X : \Omega \to \mathbb{R}$ that assigns real number to each outcome

Types:

- **Discrete**: Takes countable values
- **Continuous**: Takes uncountable values

Probability Mass Function (PMF) (discrete):

$$p_X(x) = P(X = x)$$

Properties:

- $p_X(x) \geq 0$ for all x
- $\sum_x p_X(x) = 1$

Probability Density Function (PDF) (continuous):

$$P(a \leq X \leq b) = \int_a^b f_X(x)dx$$

Properties:

- $f_X(x) \geq 0$ for all x
- $\int_{-\infty}^{\infty} f_X(x)dx = 1$
- $P(X = x) = 0$ for any specific x

Cumulative Distribution Function (CDF):

$$F_X(x) = P(X \leq x)$$

Properties:

- $0 \leq F_X(x) \leq 1$
- F_X is non-decreasing
- $\lim_{x \to -\infty} F_X(x) = 0$, $\lim_{x \to \infty} F_X(x) = 1$
- Right-continuous

Relationship to PDF: $f_X(x) = \frac{d}{dx}F_X(x)$ (where derivative exists)

Common Discrete Distributions:

1. Bernoulli Distribution $\text{Ber}(p)$:

- Single trial with success probability p
- PMF: $P(X = 1) = p$, $P(X = 0) = 1 - p$
- Mean: p, Variance: $p(1 - p)$

2. Binomial Distribution $\text{Bin}(n, p)$:

- Number of successes in n independent Bernoulli trials
- PMF: $P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$
- Mean: np, Variance: $np(1 - p)$

3. Geometric Distribution $\text{Geom}(p)$:

- Number of trials until first success
- PMF: $P(X = k) = (1 - p)^{k-1} p$ for $k = 1, 2, 3, \ldots$
- Mean: $\frac{1}{p}$, Variance: $\frac{1-p}{p^2}$
- **Memoryless property**: $P(X > m + n | X > m) = P(X > n)$

4. Poisson Distribution $\text{Pois}(\lambda)$:

- Number of events in fixed interval

- PMF: $P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$ for $k = 0, 1, 2, \ldots$
- Mean: λ, Variance: λ
- **Poisson approximation**: $\text{Bin}(n, p) \approx \text{Pois}(np)$ when n large, p small

Common Continuous Distributions:

1. Uniform Distribution $\text{Unif}(a, b)$:

- Constant density over interval $[a, b]$
- PDF: $f(x) = \frac{1}{b-a}$ for $a \leq x \leq b$, 0 otherwise
- Mean: $\frac{a+b}{2}$, Variance: $\frac{(b-a)^2}{12}$

2. Exponential Distribution $\text{Exp}(\lambda)$:

- Time between events in Poisson process
- PDF: $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$
- CDF: $F(x) = 1 - e^{-\lambda x}$ for $x \geq 0$
- Mean: $\frac{1}{\lambda}$, Variance: $\frac{1}{\lambda^2}$
- **Memoryless property**: $P(X > s + t | X > s) = P(X > t)$

3. Normal Distribution $\mathcal{N}(\mu, \sigma^2)$:

- Bell-shaped curve, most important distribution
- PDF: $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
- Mean: μ, Variance: σ^2

Standard Normal $\mathcal{N}(0, 1)$:

- PDF: $\phi(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$
- CDF: $\Phi(z) = \int_{-\infty}^{z} \phi(t) dt$

Standardization: If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $Z = \frac{X - \mu}{\sigma} \sim \mathcal{N}(0, 1)$

Expectation and Variance:

Expected Value (Mean):

Discrete: $E[X] = \sum_x x \cdot P(X = x)$

Continuous: $E[X] = \int_{-\infty}^{\infty} x \cdot f_X(x) dx$

Properties:

- **Linearity**: $E[aX + bY] = aE[X] + bE[Y]$
- **Law of Total Expectation**: $E[X] = E[E[X|Y]]$

Variance:

$$\text{Var}(X) = E[(X - E[X])^2] = E[X^2] - (E[X])^2$$

Properties:

- $\text{Var}(aX + b) = a^2\text{Var}(X)$
- If X and Y independent: $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$

Standard Deviation: $\sigma_X = \sqrt{\text{Var}(X)}$

Moment Generating Function:

$$M_X(t) = E[e^{tX}]$$

Properties:

- Uniquely determines distribution
- $M_{aX+b}(t) = e^{bt}M_X(at)$
- If X and Y independent: $M_{X+Y}(t) = M_X(t)M_Y(t)$

Joint Distributions:

Joint PMF (discrete): $p_{X,Y}(x, y) = P(X = x, Y = y)$

Joint PDF (continuous): $P((X, Y) \in A) = \iint_A f_{X,Y}(x, y)dxdy$

Marginal Distributions:

- **Discrete**: $p_X(x) = \sum_y p_{X,Y}(x, y)$
- **Continuous**: $f_X(x) = \int_{-\infty}^{\infty} f_{X,Y}(x, y)dy$

Independence: X and Y are independent if:

- **Discrete**: $p_{X,Y}(x, y) = p_X(x) \cdot p_Y(y)$ for all x, y
- **Continuous**: $f_{X,Y}(x, y) = f_X(x) \cdot f_Y(y)$ for all x, y

Covariance:

$$\text{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

Properties:

- $\text{Cov}(X, X) = \text{Var}(X)$
- $\text{Cov}(X, Y) = \text{Cov}(Y, X)$
- If X and Y independent, then $\text{Cov}(X, Y) = 0$ (converse not true)

Correlation Coefficient:

$$\rho_{X,Y} = \frac{\text{Cov}(X, Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}$$

Properties:

- $-1 \leq \rho_{X,Y} \leq 1$
- $|\rho_{X,Y}| = 1$ iff $Y = aX + b$ for some constants $a \neq 0, b$
- $\rho_{X,Y} = 0$ means uncorrelated (weaker than independence)

Limit Theorems:

Law of Large Numbers (LLN):
If $X_1, X_2, \ldots$ are i.i.d. with finite mean μ, then:

$$\frac{X_1 + X_2 + \cdots + X_n}{n} \to \mu \text{ as } n \to \infty$$

Central Limit Theorem (CLT):
If $X_1, X_2, \ldots$ are i.i.d. with mean μ and variance σ^2, then:

$$\frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}} \to \mathcal{N}(0,1) \text{ as } n \to \infty$$

where $\bar{X}_n = \frac{X_1+\cdots+X_n}{n}$

Applications:

- Approximating binomial with normal when n large
- Confidence intervals
- Hypothesis testing

Problem-Solving Tips:

For Basic Probability:

1. Identify sample space and events clearly
2. Check if events are disjoint for addition rule
3. Use complement when "at least one" appears
4. Draw tree diagrams for sequential experiments

For Conditional Probability:

1. Identify given information (condition)
2. Use definition: $P(A|B) = \frac{P(A \cap B)}{P(B)}$
3. For Bayes: identify prior, likelihood, and evidence

For Random Variables:

1. Identify distribution type from problem context
2. Use appropriate formulas for mean and variance
3. For normal distribution, standardize to use tables

GATE Tips:

- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ (inclusion-exclusion)
- Independence: $P(A \cap B) = P(A) \cdot P(B)$
- Bayes: $P(B|A) = \frac{P(A|B) \cdot P(B)}{P(A)}$
- Binomial: n trials, constant p, count successes
- Poisson: rare events, λ = rate parameter
- Normal: symmetric, bell-shaped, use standardization
- CLT: sample mean approximately normal for large n

Examples:

1. **Probability with cards**: Draw 2 cards without replacement. Find $P(\text{both aces})$.
   - $P(\text{1st ace}) = \frac{4}{52}$
   - $P(\text{2nd ace} \mid \text{1st ace}) = \frac{3}{51}$
   - $P(\text{both aces}) = \frac{4}{52} \times \frac{3}{51} = \frac{1}{221}$
2. **Bayes' theorem**: Disease affects 1% of population. Test is 95% accurate (both sensitivity and specificity). If test positive, what's probability of having disease?
   - Let D = disease, T = positive test
   - $P(D) = 0.01$, $P(T|D) = 0.95$, $P(T|D^c) = 0.05$
   - $P(T) = P(T|D)P(D) + P(T|D^c)P(D^c) = 0.95(0.01) + 0.05(0.99) = 0.0590$
   - $P(D|T) = \frac{P(T|D)P(D)}{P(T)} = \frac{0.95 \times 0.01}{0.0590} = \frac{0.0095}{0.0590} \approx 0.161$
3. **Normal distribution**: $X \sim \mathcal{N}(100, 15^2)$. Find $P(85 < X < 115)$.
   - Standardize: $Z_1 = \frac{85-100}{15} = -1$, $Z_2 = \frac{115-100}{15} = 1$
   - $P(85 < X < 115) = P(-1 < Z < 1) = \Phi(1) - \Phi(-1) = 0.8413 - 0.1587 = 0.6826$

7.2 Statistics and Hypothesis Testing (35)

Key Concepts: Statistics involves collecting, analyzing, and interpreting data. Hypothesis testing provides framework for making decisions under uncertainty using sample data.

Descriptive Statistics:

Measures of Central Tendency:

Sample Mean: $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$

Sample Median: Middle value when data ordered

- If n odd: median = $x_{(n+1)/2}$
- If n even: median = $\frac{x_{n/2} + x_{n/2+1}}{2}$

Sample Mode: Most frequently occurring value

Properties:

- Mean affected by outliers, median robust

- For symmetric distributions: mean ≈ median
- For right-skewed: mean > median
- For left-skewed: mean < median

Measures of Variability:

Sample Variance: $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$

Sample Standard Deviation: $s = \sqrt{s^2}$

Range: $\max(x_i) - \min(x_i)$

Interquartile Range (IQR): $Q_3 - Q_1$

- Q_1: 25th percentile, Q_3: 75th percentile
- Robust to outliers

Coefficient of Variation: $CV = \frac{s}{\bar{x}}$ (relative variability)

Sampling Distributions:

Population vs Sample:

- **Population**: Entire group of interest
- **Sample**: Subset of population
- **Parameter**: Population characteristic (μ, σ, p)
- **Statistic**: Sample characteristic ($\bar{x}$, s, $\hat{p}$)

Sampling Distribution: Distribution of sample statistic across all possible samples

Central Limit Theorem for Sample Mean:
If $X_1, \ldots, X_n$ are i.i.d. with mean μ and variance σ^2, then:

$$\bar{X} \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right) \text{ approximately for large } n$$

Standard Error: $SE(\bar{X}) = \frac{\sigma}{\sqrt{n}}$

Key Distributions for Inference:

1. Standard Normal Distribution $\mathcal{N}(0,1)$:

- Used when σ known or n large
- $Z = \frac{\bar{X} - \mu}{\sigma/\sqrt{n}}$

2. t-Distribution t_ν:

- Used when σ unknown and n small
- Parameter: degrees of freedom $\nu = n - 1$

- $T = \frac{\bar{X}-\mu}{S/\sqrt{n}}$ where S is sample standard deviation
- Approaches standard normal as $\nu \to \infty$
- Heavier tails than normal

3. Chi-Square Distribution χ^2_ν:

- Used for variance testing and goodness-of-fit
- $\chi^2 = \frac{(n-1)S^2}{\sigma^2}$ has χ^2_{n-1} distribution
- Right-skewed, non-negative

4. F-Distribution F_{ν_1,ν_2}:

- Used for comparing variances, ANOVA
- Ratio of two chi-square random variables
- Parameters: numerator df ν_1, denominator df ν_2

Point Estimation:

Estimator: Function of sample data used to estimate parameter

- $\hat{\theta} = g(X_1, \ldots, X_n)$

Properties of Estimators:

1. Unbiasedness: $E[\hat{\theta}] = \theta$

- Sample mean is unbiased for population mean
- Sample variance s^2 is unbiased for σ^2 (uses $n-1$ denominator)

2. Consistency: $\hat{\theta}_n \to \theta$ as $n \to \infty$

3. Efficiency: Among unbiased estimators, one with smallest variance

Method of Moments: Set sample moments equal to population moments

- For normal distribution: $\hat{\mu} = \bar{X}$, $\hat{\sigma}^2 = S^2$

Maximum Likelihood Estimation (MLE):

- Choose parameter value that maximizes likelihood of observed data
- Often involves solving $\frac{d}{d\theta}\ln L(\theta) = 0$

Interval Estimation:

Confidence Interval: Range of plausible values for parameter

Interpretation: If we construct 95% confidence intervals repeatedly, 95% will contain true parameter

Confidence Interval for Mean:

Case 1: σ known or $n \geq 30$

$$\bar{x} \pm z_{\alpha/2}\frac{\sigma}{\sqrt{n}}$$

Case 2: σ unknown and $n < 30$ (assume normal population)

$$\bar{x} \pm t_{\alpha/2,n-1}\frac{s}{\sqrt{n}}$$

Confidence Interval for Proportion:

$$\hat{p} \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

where $\hat{p} = \frac{x}{n}$ (sample proportion)

Confidence Interval for Variance:

$$\frac{(n-1)s^2}{\chi^2_{\alpha/2,n-1}} \leq \sigma^2 \leq \frac{(n-1)s^2}{\chi^2_{1-\alpha/2,n-1}}$$

Factors Affecting Interval Width:

- **Confidence level**: Higher confidence → wider interval
- **Sample size**: Larger n → narrower interval
- **Population variability**: Larger σ → wider interval

Hypothesis Testing:

Components:

- **Null hypothesis** H_0: Status quo, no effect
- **Alternative hypothesis** H_1 (or H_a): What we want to prove
- **Test statistic**: Function of sample data
- **Critical region**: Values leading to rejection of H_0
- **Significance level** α: Probability of Type I error

Types of Tests:

- **Two-tailed**: $H_1 : \theta \neq \theta_0$
- **Right-tailed**: $H_1 : \theta > \theta_0$
- **Left-tailed**: $H_1 : \theta < \theta_0$

Types of Errors:

- **Type I Error**: Reject true H_0 (probability = α)
- **Type II Error**: Fail to reject false H_0 (probability = β)

- **Power**: $1 - \beta$ (probability of correctly rejecting false H_0)

Testing Procedure:

1. State hypotheses
2. Choose significance level α
3. Calculate test statistic
4. Find p-value or critical value
5. Make decision and interpret

Tests for One Sample:

1. Test for Mean (σ known):

- Test statistic: $Z = \frac{\bar{X} - \mu_0}{\sigma/\sqrt{n}}$
- Reject H_0 if $|Z| > z_{\alpha/2}$ (two-tailed)

2. Test for Mean (σ unknown):

- Test statistic: $T = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}$
- Reject H_0 if $|T| > t_{\alpha/2, n-1}$ (two-tailed)

3. Test for Proportion:

- Test statistic: $Z = \frac{\hat{p} - p_0}{\sqrt{p_0(1-p_0)/n}}$
- Reject H_0 if $|Z| > z_{\alpha/2}$ (two-tailed)

4. Test for Variance:

- Test statistic: $\chi^2 = \frac{(n-1)S^2}{\sigma_0^2}$
- Reject H_0 if $\chi^2 > \chi^2_{\alpha/2, n-1}$ or $\chi^2 < \chi^2_{1-\alpha/2, n-1}$ (two-tailed)

Tests for Two Samples:

1. Comparison of Two Means (independent samples):

Case 1: σ_1, σ_2 known

- Test statistic: $Z = \frac{(\bar{X}_1 - \bar{X}_2) - (\mu_1 - \mu_2)}{\sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}}$

Case 2: $\sigma_1 = \sigma_2 = \sigma$ (unknown)

- Pooled variance: $s_p^2 = \frac{(n_1-1)s_1^2 + (n_2-1)s_2^2}{n_1+n_2-2}$
- Test statistic: $T = \frac{(\bar{X}_1 - \bar{X}_2) - (\mu_1 - \mu_2)}{s_p\sqrt{1/n_1 + 1/n_2}}$
- df = $n_1 + n_2 - 2$

Case 3: $\sigma_1 \neq \sigma_2$ (Welch's t-test)

- Test statistic: $T = \frac{(\bar{X}_1 - \bar{X}_2) - (\mu_1 - \mu_2)}{\sqrt{s_1^2/n_1 + s_2^2/n_2}}$
- df = $\frac{(s_1^2/n_1 + s_2^2/n_2)^2}{(s_1^2/n_1)^2/(n_1-1) + (s_2^2/n_2)^2/(n_2-1)}$

2. Paired t-test:

- For dependent samples (before/after, matched pairs)
- Test statistic: $T = \frac{\bar{D} - \mu_D}{S_D/\sqrt{n}}$
- where $D_i = X_i - Y_i$ are differences

3. Comparison of Two Proportions:

- Pooled proportion: $\hat{p} = \frac{x_1 + x_2}{n_1 + n_2}$
- Test statistic: $Z = \frac{\hat{p}_1 - \hat{p}_2}{\sqrt{\hat{p}(1-\hat{p})(1/n_1 + 1/n_2)}}$

4. F-test for Equality of Variances:

- Test statistic: $F = \frac{S_1^2}{S_2^2}$ (larger variance in numerator)
- df = $(n_1 - 1, n_2 - 1)$

Chi-Square Tests:

1. Goodness of Fit Test:

- Tests if sample follows specified distribution
- Test statistic: $\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$
- where O_i = observed frequency, E_i = expected frequency
- df = $k - 1 - \text{number of estimated parameters}$

2. Test of Independence:

- Tests if two categorical variables are independent
- Test statistic: $\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$
- where $E_{ij} = \frac{\text{row}_i \text{ total} \times \text{column}_j \text{ total}}{\text{grand total}}$
- df = $(r - 1)(c - 1)$

Analysis of Variance (ANOVA):

One-Way ANOVA: Compare means of k groups

- H_0 : $\mu_1 = \mu_2 = \cdots = \mu_k$
- H_1: At least one mean differs

Test Statistic: $F = \frac{MST}{MSE} = \frac{SST/(k-1)}{SSE/(n-k)}$

where:

- $SST = \sum_{i=1}^{k} n_i(\bar{x}_i - \bar{x})^2$ (between groups)
- $SSE = \sum_{i=1}^{k} \sum_{j=1}^{n_i}(x_{ij} - \bar{x}_i)^2$ (within groups)
- $MST = SST/(k-1)$, $MSE = SSE/(n-k)$

Assumptions:

- Normality within each group
- Equal variances (homoscedasticity)
- Independence of observations

Regression Analysis:

Simple Linear Regression: $Y = \beta_0 + \beta_1 X + \epsilon$

Least Squares Estimates:

- $\hat{\beta}_1 = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$
- $\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$

Coefficient of Determination: $R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$

- Proportion of variance explained by regression

Testing Significance of Regression:

- $H_0 : \beta_1 = 0$ vs $H_1 : \beta_1 \neq 0$
- Test statistic: $T = \frac{\hat{\beta}_1}{SE(\hat{\beta}_1)}$

Non-parametric Tests:

1. Sign Test: Tests median of population

- Uses only signs of differences from hypothesized median
- Distribution-free

2. Wilcoxon Signed-Rank Test: Tests median (uses ranks)

- More powerful than sign test
- Assumes symmetric distribution

3. Mann-Whitney U Test: Compares two independent samples

- Non-parametric alternative to two-sample t-test
- Tests if populations have same distribution

4. Kruskal-Wallis Test: Compares multiple groups

- Non-parametric alternative to one-way ANOVA

Problem-Solving Tips:

For Confidence Intervals:

1. Identify parameter of interest
2. Check conditions (normality, sample size)
3. Choose appropriate distribution (z, t, χ^2)
4. Calculate margin of error
5. Interpret in context

For Hypothesis Testing:

1. State hypotheses clearly
2. Check assumptions
3. Calculate test statistic
4. Find p-value or compare to critical value
5. Make decision and interpret in context

GATE Tips:

- Use t-distribution when σ unknown and n < 30
- Chi-square for variance tests and categorical data
- F-distribution for comparing variances and ANOVA
- p-value < α $\implies$ reject H_0
- Type I error = α, Type II error = β, Power = 1-β
- Larger sample size $\implies$ narrower confidence interval
- ANOVA tests equality of means across groups
- R^2 measures proportion of variance explained

Examples:

1. **Confidence interval for mean**: Sample of 25 students has mean score 78 with standard deviation 12. Find 95% CI for population mean.
   - Use t-distribution: $n = 25 < 30$, σ unknown
   - $t_{0.025,24} = 2.064$
   - CI: $78 \pm 2.064 \times \frac{12}{\sqrt{25}} = 78 \pm 4.95 = (73.05, 82.95)$
2. **Hypothesis test for proportion**: Company claims 90% customer satisfaction. Sample of 200 shows 170 satisfied. Test at α = 0.05.
   - $H_0 : p = 0.9$ vs $H_1 : p \neq 0.9$
   - $\hat{p} = 170/200 = 0.85$
   - $Z = \frac{0.85-0.9}{\sqrt{0.9 \times 0.1/200}} = \frac{-0.05}{0.0212} = -2.36$
   - p-value = $2 \times P(Z < -2.36) = 2 \times 0.0091 = 0.0182$

- Since p-value < 0.05, reject H_0

3. **Chi-square goodness of fit**: Test if die is fair. Observed frequencies: 1:8, 2:12, 3:10, 4:15, 5:9, 6:6 (total 60 rolls).
   - Expected frequency for each face: 60/6 = 10
   - $\chi^2 = \frac{(8-10)^2}{10} + \frac{(12-10)^2}{10} + \cdots + \frac{(6-10)^2}{10} = 0.4 + 0.4 + 0 + 2.5 + 0.1 + 1.6 = 5.0$
   - df = 6 - 1 = 5
   - Critical value at α = 0.05: $\chi^2_{0.05,5} = 11.07$
   - Since 5.0 < 11.07, fail to reject H_0 (die appears fair)

7.3 Regression Analysis (25)

Key Concepts: Regression analysis studies relationships between variables, allowing prediction and understanding of how one variable affects another. It's fundamental in data science, economics, and engineering.

Simple Linear Regression:

Model: $Y = \beta_0 + \beta_1 X + \epsilon$

where:

- Y: dependent (response) variable
- X: independent (predictor) variable
- β_0: y-intercept (value of Y when X = 0)
- β_1: slope (change in Y per unit change in X)
- ϵ: random error term

Assumptions:

1. **Linearity**: Relationship between X and Y is linear
2. **Independence**: Observations are independent
3. **Homoscedasticity**: Constant variance of errors
4. **Normality**: Errors are normally distributed
5. **No measurement error** in X

Least Squares Estimation:

Objective: Minimize sum of squared residuals

$$\text{SSE} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$$

Normal Equations:

$$\frac{\partial \text{SSE}}{\partial \beta_0} = 0, \quad \frac{\partial \text{SSE}}{\partial \beta_1} = 0$$

Solutions:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} = \frac{S_{xy}}{S_{xx}}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

where:

- $S_{xy} = \sum(x_i - \bar{x})(y_i - \bar{y})$ (sum of cross-products)
- $S_{xx} = \sum(x_i - \bar{x})^2$ (sum of squares for X)
- $S_{yy} = \sum(y_i - \bar{y})^2$ (sum of squares for Y)

Alternative Formulas:

$$\hat{\beta}_1 = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$$

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum x_i}{n}$$

Fitted Values and Residuals:

Fitted value: $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$

Residual: $e_i = y_i - \hat{y}_i$ (observed - predicted)

Properties of Residuals:

- $\sum e_i = 0$ (sum to zero)
- $\sum x_i e_i = 0$ (uncorrelated with X)
- $\sum \hat{y}_i e_i = 0$ (uncorrelated with fitted values)

Analysis of Variance (ANOVA) for Regression:

Total Sum of Squares: $\text{SST} = \sum(y_i - \bar{y})^2$ (total variation)

Regression Sum of Squares: $\text{SSR} = \sum(\hat{y}_i - \bar{y})^2$ (explained variation)

Error Sum of Squares: $\text{SSE} = \sum(y_i - \hat{y}_i)^2$ (unexplained variation)

Fundamental Identity: $\text{SST} = \text{SSR} + \text{SSE}$

Degrees of Freedom:

- SST: $n - 1$
- SSR: 1 (one predictor)
- SSE: $n - 2$

Mean Squares:

- $\text{MSR} = \text{SSR}/1$
- $\text{MSE} = \text{SSE}/(n-2)$

Coefficient of Determination:

$$R^2 = \frac{\text{SSR}}{\text{SST}} = 1 - \frac{\text{SSE}}{\text{SST}}$$

Interpretation: Proportion of total variation in Y explained by X

- $0 \leq R^2 \leq 1$
- $R^2 = 1$: Perfect fit
- $R^2 = 0$: No linear relationship

Relationship to Correlation: $R^2 = r_{xy}^2$ where r_{xy} is sample correlation coefficient

Statistical Inference:

Standard Error of Regression: $s = \sqrt{\text{MSE}} = \sqrt{\frac{\text{SSE}}{n-2}}$

Standard Errors of Coefficients:

$$SE(\hat{\beta}_1) = \frac{s}{\sqrt{S_{xx}}}, \quad SE(\hat{\beta}_0) = s\sqrt{\frac{1}{n} + \frac{\bar{x}^2}{S_{xx}}}$$

Confidence Intervals:

- For β_1: $\hat{\beta}_1 \pm t_{\alpha/2,n-2} \cdot SE(\hat{\beta}_1)$
- For β_0: $\hat{\beta}_0 \pm t_{\alpha/2,n-2} \cdot SE(\hat{\beta}_0)$

Hypothesis Tests:

Test for Slope:

- $H_0 : \beta_1 = 0$ vs $H_1 : \beta_1 \neq 0$
- Test statistic: $t = \frac{\hat{\beta}_1}{SE(\hat{\beta}_1)}$
- Reject if $|t| > t_{\alpha/2,n-2}$

F-test for Overall Regression:

- $H_0 : \beta_1 = 0$ vs $H_1 : \beta_1 \neq 0$
- Test statistic: $F = \frac{\text{MSR}}{\text{MSE}}$
- Reject if $F > F_{\alpha,1,n-2}$
- Note: $F = t^2$ for simple linear regression

Prediction:

Point Prediction: $\hat{y}_0 = \hat{\beta}_0 + \hat{\beta}_1 x_0$

Prediction Interval (for new observation):

$$\hat{y}_0 \pm t_{\alpha/2,n-2} \cdot s\sqrt{1 + \frac{1}{n} + \frac{(x_0 - \bar{x})^2}{S_{xx}}}$$

Confidence Interval for Mean Response:

$$\hat{y}_0 \pm t_{\alpha/2,n-2} \cdot s\sqrt{\frac{1}{n} + \frac{(x_0 - \bar{x})^2}{S_{xx}}}$$

Note: Prediction interval is wider than confidence interval

Multiple Linear Regression:

Model: $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_k X_k + \epsilon$

Matrix Form: $\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$

where:

- $\mathbf{Y}$: $n \times 1$ response vector
- $\mathbf{X}$: $n \times (k+1)$ design matrix
- $\boldsymbol{\beta}$: $(k+1) \times 1$ parameter vector
- $\boldsymbol{\epsilon}$: $n \times 1$ error vector

Least Squares Solution:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

Properties:

- Unbiased: $E[\hat{\boldsymbol{\beta}}] = \boldsymbol{\beta}$
- Covariance: $\text{Cov}(\hat{\boldsymbol{\beta}}) = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$

ANOVA for Multiple Regression:

Degrees of Freedom:

- SSR: k (number of predictors)
- SSE: $n - k - 1$
- SST: $n - 1$

F-test for Overall Model:

- $H_0: \beta_1 = \beta_2 = \cdots = \beta_k = 0$
- Test statistic: $F = \frac{\text{MSR}}{\text{MSE}} = \frac{\text{SSR}/k}{\text{SSE}/(n-k-1)}$

Adjusted R-squared:

$$R^2_{adj} = 1 - \frac{\text{SSE}/(n-k-1)}{\text{SST}/(n-1)} = 1 - (1-R^2)\frac{n-1}{n-k-1}$$

Purpose: Penalizes for adding variables that don't improve fit significantly

Individual t-tests:

- $H_0 : \beta_j = 0$ vs $H_1 : \beta_j \neq 0$
- Test statistic: $t = \frac{\hat{\beta}_j}{SE(\hat{\beta}_j)}$

Multicollinearity:

Problem: High correlation among predictor variables

- Makes coefficient estimates unstable
- Inflates standard errors
- Doesn't affect prediction accuracy

Detection:

- **Correlation matrix**: High pairwise correlations (|r| > 0.8)
- **Variance Inflation Factor (VIF)**: $\text{VIF}_j = \frac{1}{1-R^2_j}$
  where R^2_j is R² from regressing X_j on other predictors
- VIF > 10 indicates serious multicollinearity

Solutions:

- Remove highly correlated variables
- Principal component regression
- Ridge regression

Model Selection:

Forward Selection: Start with no variables, add significant ones

Backward Elimination: Start with all variables, remove non-significant ones

Stepwise Regression: Combination of forward and backward

Information Criteria:

- **AIC**: $\text{AIC} = n\ln(\text{SSE}/n) + 2(k+1)$
- **BIC**: $\text{BIC} = n\ln(\text{SSE}/n) + (k+1)\ln(n)$
- Lower values indicate better models

Cross-Validation: Split data into training and validation sets

Residual Analysis:

Purposes:

- Check model assumptions
- Identify outliers and influential points
- Suggest model improvements

Diagnostic Plots:

1. **Residuals vs Fitted Values**:
   - Check linearity and homoscedasticity
   - Should show random scatter around zero
2. **Normal Q-Q Plot**:
   - Check normality of residuals
   - Points should lie approximately on straight line
3. **Residuals vs Predictor Variables**:
   - Check for non-linear relationships
   - May suggest transformations
4. **Residuals vs Order**:
   - Check independence assumption
   - Look for patterns or trends

Standardized Residuals:

$$r_i = \frac{e_i}{s\sqrt{1 - h_{ii}}}$$

where h_{ii} is leverage (diagonal element of hat matrix)

Outliers and Influential Points:

Outlier: Observation with large residual

- Standardized residual $|r_i| > 2$ or 3

High Leverage Point: Observation with unusual X values

- Leverage $h_{ii} > 2(k+1)/n$

Influential Point: Observation that greatly affects fitted model

- **Cook's Distance**: $D_i = \frac{r_i^2}{k+1} \cdot \frac{h_{ii}}{1-h_{ii}}$
- $D_i > 1$ indicates influential point

Transformations:

Box-Cox Transformation: $Y^{(\lambda)} = \frac{Y^\lambda - 1}{\lambda}$ (λ ≠ 0)

- λ = 1: No transformation

- λ = 0: Log transformation
- λ = 0.5: Square root transformation
- λ = -1: Reciprocal transformation

Polynomial Regression:

Model: $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_k X^k + \epsilon$

Considerations:

- Higher-order terms can cause overfitting
- Extrapolation beyond data range dangerous
- Orthogonal polynomials reduce multicollinearity

Logistic Regression:

Model: For binary response (0/1)

$$\ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k$$

where $p = P(Y = 1)$

Logistic Function: $p = \frac{e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k}}{1 + e^{\beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k}}$

Estimation: Maximum likelihood (not least squares)

Problem-Solving Tips:

For Simple Linear Regression:

1. Plot data to check linearity
2. Calculate slope and intercept using formulas
3. Compute R² to assess fit quality
4. Check residual plots for assumption violations

For Multiple Regression:

1. Check for multicollinearity (VIF)
2. Use F-test for overall model significance
3. Use t-tests for individual coefficients
4. Consider model selection techniques

For Model Diagnostics:

1. Always plot residuals vs fitted values
2. Check normal Q-Q plot of residuals
3. Look for outliers and influential points

4. Consider transformations if assumptions violated

GATE Tips:

- Least squares minimizes sum of squared residuals
- R² = proportion of variance explained (0 to 1)
- Higher R² doesn't always mean better model
- Check assumptions through residual analysis
- Multicollinearity affects coefficient stability, not predictions
- F-test for overall model, t-tests for individual coefficients
- Prediction intervals wider than confidence intervals
- Adjusted R² penalizes for additional variables

Examples:

1. **Simple linear regression**: Data points (1,2), (2,4), (3,5), (4,7), (5,8)
   - $\bar{x} = 3, \bar{y} = 5.2$
   - $S_{xx} = \sum(x_i - \bar{x})^2 = 10$
   - $S_{xy} = \sum(x_i - \bar{x})(y_i - \bar{y}) = 11.2$
   - $\hat{\beta}_1 = 11.2/10 = 1.12$
   - $\hat{\beta}_0 = 5.2 - 1.12(3) = 1.84$
   - Equation: $\hat{y} = 1.84 + 1.12x$
2. **R² calculation**:
   - $\text{SST} = \sum(y_i - \bar{y})^2 = 16.8$
   - $\text{SSR} = \hat{\beta}_1^2 S_{xx} = (1.12)^2(10) = 12.544$
   - $R^2 = 12.544/16.8 = 0.747$
   - 74.7% of variation in Y explained by X
3. **Hypothesis test for slope**:
   - $\text{SSE} = \text{SST} - \text{SSR} = 16.8 - 12.544 = 4.256$
   - $s = \sqrt{4.256/3} = 1.191$
   - $SE(\hat{\beta}_1) = 1.191/\sqrt{10} = 0.377$
   - $t = 1.12/0.377 = 2.97$
   - With df = 3, critical value $t_{0.025,3} = 3.182$
   - Since $|2.97| < 3.182$, fail to reject $H_0 : \beta_1 = 0$ at α = 0.05

8. General Aptitude (200 Questions)

8.1 Verbal Ability (100)

Key Concepts: Verbal ability tests comprehension, vocabulary, grammar, and communication skills essential for technical professionals.

Reading Comprehension (25):

Types of Passages:

- **Factual**: Presents information, data, or research findings
- **Analytical**: Examines cause-effect relationships, comparisons
- **Argumentative**: Presents viewpoint with supporting evidence
- **Narrative**: Tells story or describes sequence of events

Question Types:

1. Main Idea Questions:

- "The primary purpose of the passage is..."
- "The passage is primarily concerned with..."
- **Strategy**: Look for thesis statement, topic sentences, concluding remarks

2. Detail Questions:

- "According to the passage..."
- "The author mentions X in order to..."
- **Strategy**: Locate specific information, avoid inferences

3. Inference Questions:

- "It can be inferred that..."
- "The passage suggests that..."
- **Strategy**: Draw logical conclusions from stated information

4. Tone/Attitude Questions:

- "The author's attitude toward X is..."
- "The tone of the passage is..."
- **Options**: Critical, supportive, neutral, skeptical, optimistic, pessimistic

5. Vocabulary in Context:

- "In line X, 'word' most nearly means..."
- **Strategy**: Consider surrounding context, not just dictionary meaning

6. Structure Questions:

- "The organization of the passage is..."
- "The author develops the argument by..."

Reading Strategies:

Active Reading:

1. **Preview**: Scan title, first/last paragraphs, topic sentences

2. **Purpose**: Identify why you're reading (main idea, specific detail)
3. **Predict**: Anticipate content based on preview
4. **Read**: Focus on key ideas, not every detail
5. **Review**: Summarize main points

Time Management:

- Spend 2-3 minutes reading passage
- 1-2 minutes per question
- Don't get stuck on difficult questions

Common Traps:

- **Extreme language**: "always," "never," "all," "none"
- **Out of scope**: Information not in passage
- **Opposite answers**: Contradicts passage information
- **Distortion**: Twists passage information

Grammar and Usage (25):

Subject-Verb Agreement:

Basic Rule: Singular subjects take singular verbs; plural subjects take plural verbs

Tricky Cases:

- **Collective nouns**: "The team is/are playing" (depends on context)
- **Compound subjects**: "John and Mary are..." (plural)
- **Either/or, neither/nor**: Verb agrees with nearest subject
- **Indefinite pronouns**:
    - Singular: each, every, either, neither, one, someone, anyone, everyone
    - Plural: both, few, many, several
    - Variable: all, some, most, none (depends on object)

Examples:

- "Each of the students has submitted their assignment." (Incorrect - pronoun disagreement)
- "Each of the students has submitted his or her assignment." (Correct)
- "Neither the teacher nor the students were ready." (Correct - agrees with "students")

Pronoun Usage:

Pronoun-Antecedent Agreement:

- Pronoun must agree with antecedent in number, gender, person

- "Every student must bring their book." (Incorrect)
- "Every student must bring his or her book." (Correct)

Pronoun Case:

- **Subjective**: I, you, he, she, it, we, they
- **Objective**: me, you, him, her, it, us, them
- **Possessive**: my, your, his, her, its, our, their

Common Errors:

- "Between you and I" (Incorrect - should be "me")
- "Who/Whom": Who = subject, Whom = object
- "Its/It's": Its = possessive, It's = it is

Verb Tenses:

Simple Tenses:

- **Present**: "I write"
- **Past**: "I wrote"
- **Future**: "I will write"

Perfect Tenses:

- **Present Perfect**: "I have written" (completed action with present relevance)
- **Past Perfect**: "I had written" (completed before another past action)
- **Future Perfect**: "I will have written" (will be completed by future time)

Progressive Tenses:

- **Present Progressive**: "I am writing" (ongoing now)
- **Past Progressive**: "I was writing" (ongoing in past)
- **Future Progressive**: "I will be writing" (ongoing in future)

Sequence of Tenses:

- Main clause past → subordinate clause past
- "He said that he was tired." (Not "He said that he is tired.")

Modifiers:

Misplaced Modifiers:

- "I saw a man with binoculars walking down the street." (Ambiguous)
- "Walking down the street, I saw a man with binoculars." (Clear)

Dangling Modifiers:

- "Having finished the assignment, the TV was turned on." (Incorrect)
- "Having finished the assignment, I turned on the TV." (Correct)

Parallel Structure:

Lists: Items in series should have same grammatical form

- "I like reading, writing, and to swim." (Incorrect)
- "I like reading, writing, and swimming." (Correct)

Correlative Conjunctions: Both...and, either...or, neither...nor, not only...but also

- "She is not only intelligent but also has creativity." (Incorrect)
- "She is not only intelligent but also creative." (Correct)

Sentence Structure:

Fragments: Incomplete sentences missing subject or verb

- "Because I was tired." (Fragment)
- "I went home because I was tired." (Complete)

Run-on Sentences: Two or more independent clauses incorrectly joined

- "I was tired I went home." (Run-on)
- "I was tired, so I went home." (Correct)

Comma Splices: Two independent clauses joined only by comma

- "I was tired, I went home." (Comma splice)
- "I was tired; I went home." (Correct)

Vocabulary (25):

Word Roots, Prefixes, Suffixes:

Common Prefixes:

- **Anti-**: against (antibiotic, antisocial)
- **Pre-**: before (preview, prehistoric)
- **Post-**: after (postwar, postpone)
- **Sub-**: under (submarine, substandard)
- **Super-**: above (superhuman, supervisor)
- **Inter-**: between (international, interact)
- **Intra-**: within (intramural, intravenous)
- **Extra-**: beyond (extraordinary, extraterrestrial)

Common Suffixes:

- **-tion/-sion**: action/state (creation, decision)
- **-ment**: result/state (development, agreement)
- **-ness**: quality (happiness, darkness)
- **-ful**: full of (helpful, beautiful)
- **-less**: without (hopeless, careless)
- **-able/-ible**: capable of (readable, visible)

Common Roots:

- **Bene-**: good (benefit, benevolent)
- **Mal-**: bad (malfunction, malicious)
- **Chron-**: time (chronology, chronic)
- **Graph-**: write (biography, telegraph)
- **Phon-**: sound (telephone, phonics)
- **Geo-**: earth (geography, geology)

Context Clues:

Definition: Word meaning explained in sentence

- "The edifice, a large imposing building, dominated the skyline."

Example: Specific instances given

- "Citrus fruits, such as oranges and lemons, are rich in vitamin C."

Contrast: Opposite meaning indicated

- "Unlike his gregarious brother, Tom was quite introverted."

Cause and Effect: Relationship shows meaning

- "The drought caused the crops to desiccate and wither away."

Synonyms and Antonyms:

High-Frequency GATE Words:

Positive Connotation:

- Acclaim, commend, extol, laud, praise
- Abundant, copious, plentiful, profuse
- Astute, discerning, perceptive, shrewd
- Benevolent, charitable, magnanimous, philanthropic

Negative Connotation:

- Censure, condemn, denounce, reproach

- Meager, paltry, scanty, sparse
- Gullible, naive, credulous
- Malevolent, vindictive, spiteful

Neutral/Technical:

- Analyze, scrutinize, examine, investigate
- Hypothesis, theory, conjecture, postulate
- Implement, execute, accomplish, achieve
- Significant, substantial, considerable, notable

Analogies:

Common Relationship Types:

1. Synonyms: HAPPY : JOYFUL
2. Antonyms: HOT : COLD
3. Part to Whole: WHEEL : CAR
4. Cause to Effect: RAIN : FLOOD
5. Function: SCISSORS : CUT
6. Category: ROSE : FLOWER
7. Degree: WARM : HOT
8. Location: BOOK : LIBRARY
9. Worker to Tool: PAINTER : BRUSH
10. Characteristic: ICE : COLD

Strategy:

1. Identify relationship between first pair
2. Look for same relationship in answer choices
3. Make sentence: "A is to B as C is to D"
4. Check if sentence makes sense

Sentence Completion (25):

Types of Clues:

1. Definition/Restatement:

- Signal words: that is, in other words, namely
- "The professor's lecture was so _ that students fell asleep; in other words, it was extremely boring."

2. Contrast/Opposition:

- Signal words: but, however, although, despite, nevertheless
- "Although she appeared confident, she was actually quite _."

3. Cause and Effect:

- Signal words: because, since, therefore, consequently, as a result
- "Because of the severe drought, the harvest was _."

4. Examples:

- Signal words: such as, for example, including
- "The museum displayed various artifacts, including _ pottery and ancient tools."

5. Comparison:

- Signal words: like, similar to, just as
- "Like his mentor, the young scientist was _ in his research methods."

Strategy:

1. Read entire sentence for context
2. Identify signal words and clue types
3. Predict word before looking at choices
4. Eliminate obviously wrong answers
5. Check answer in context

Common Patterns:

- **Positive/Negative**: Look for words that indicate tone
- **Degree**: Mild vs. extreme words
- **Time sequence**: Before/after relationships
- **Logic flow**: Cause leads to logical effect

Problem-Solving Tips:

For Reading Comprehension:

1. Read questions first to know what to look for
2. Focus on first and last paragraphs for main ideas
3. Pay attention to transition words (however, therefore, moreover)
4. Eliminate extreme or absolute answer choices
5. Stay within scope of passage

For Grammar:

1. Read sentence aloud to catch errors
2. Identify subject and verb first
3. Check for parallel structure in lists
4. Watch for pronoun-antecedent agreement

5. Be careful with modifier placement

For Vocabulary:

1. Use word roots to deduce meanings
2. Consider context clues carefully
3. Eliminate choices that don't fit context
4. Don't choose words just because they sound sophisticated
5. Practice with high-frequency GATE vocabulary

GATE Tips:

- Time management crucial: don't spend too long on any question
- Process of elimination often more effective than direct selection
- Read all answer choices before selecting
- Trust first instinct if unsure
- Practice regularly with previous years' questions
- Focus on commonly tested grammar rules
- Build vocabulary through reading and word lists
- Pay attention to question types and develop specific strategies

Common Mistakes to Avoid:

- Overthinking simple questions
- Choosing answers based on partial information
- Ignoring context clues in vocabulary questions
- Misreading question stems
- Not checking answers in context
- Spending too much time on difficult passages
- Confusing similar-looking answer choices

8.2 Quantitative Aptitude (100)

Key Concepts: Quantitative aptitude tests mathematical reasoning, numerical ability, and problem-solving skills using basic mathematical concepts.

Arithmetic (30):

Number Systems:

Natural Numbers: 1, 2, 3, 4, ... (counting numbers)
Whole Numbers: 0, 1, 2, 3, 4, ... (natural numbers + 0)
Integers: ..., -2, -1, 0, 1, 2, ... (positive and negative whole numbers)
Rational Numbers: Numbers that can be expressed as p/q where p, q are integers, $q \neq 0$

Irrational Numbers: Cannot be expressed as fraction (√2, π, e)
Real Numbers: All rational and irrational numbers

Properties of Numbers:

Even Numbers: Divisible by 2 (2, 4, 6, 8, ...)
Odd Numbers: Not divisible by 2 (1, 3, 5, 7, ...)

Operations:

- Even ± Even = Even
- Odd ± Odd = Even
- Even ± Odd = Odd
- Even × Even = Even
- Odd × Odd = Odd
- Even × Odd = Even

Prime Numbers: Natural numbers > 1 with exactly two factors (1 and itself)

- First few primes: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47...
- 2 is only even prime number
- All primes > 2 are odd

Composite Numbers: Natural numbers > 1 that are not prime

- Have more than two factors
- Can be expressed as product of primes (fundamental theorem of arithmetic)

Divisibility Rules:

- **2**: Last digit is even (0, 2, 4, 6, 8)
- **3**: Sum of digits divisible by 3
- **4**: Last two digits divisible by 4
- **5**: Last digit is 0 or 5
- **6**: Divisible by both 2 and 3
- **8**: Last three digits divisible by 8
- **9**: Sum of digits divisible by 9
- **10**: Last digit is 0
- **11**: Alternating sum of digits divisible by 11

HCF and LCM:

Highest Common Factor (HCF): Largest number that divides all given numbers
Least Common Multiple (LCM): Smallest number divisible by all given numbers

Methods:

1. **Prime Factorization**:
   - HCF = Product of lowest powers of common prime factors
   - LCM = Product of highest powers of all prime factors
2. **Euclidean Algorithm** (for HCF):
   - HCF(a,b) = HCF(b, a mod b)
   - Continue until remainder is 0

Relationship: For two numbers a and b:

$$a \times b = \text{HCF}(a, b) \times \text{LCM}(a, b)$$

Fractions:

Types:

- **Proper fraction**: Numerator < Denominator (3/5)
- **Improper fraction**: Numerator ≥ Denominator (7/5)
- **Mixed number**: Whole number + proper fraction (1 2/5)

Operations:

- **Addition/Subtraction**: Find common denominator

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd}$$

- **Multiplication**: Multiply numerators and denominators

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

- **Division**: Multiply by reciprocal

$$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \times \frac{d}{c} = \frac{ad}{bc}$$

Decimals:

Types:

- **Terminating**: Finite decimal places (0.25, 0.125)
- **Non-terminating recurring**: Infinite decimal places with repeating pattern (0.333..., 0.142857142857...)
- **Non-terminating non-recurring**: Infinite non-repeating (π, √2)

Conversion:

- **Fraction to Decimal**: Divide numerator by denominator
- **Decimal to Fraction**:

- 0.25 = 25/100 = 1/4
- 0.333... = 1/3
- 0.142857... = 1/7

Percentages:

Basic Concepts:

- Percent means "per hundred"
- 25% = 25/100 = 0.25 = 1/4

Important Conversions:

- 1/2 = 50%, 1/3 = 33.33%, 1/4 = 25%, 1/5 = 20%
- 1/6 = 16.67%, 1/8 = 12.5%, 1/10 = 10%

Percentage Change:

$$\text{Percentage Change} = \frac{\text{New Value - Old Value}}{\text{Old Value}} \times 100\%$$

Successive Percentage Changes:
If quantity changes by a% then b%:

$$\text{Net Change} = a + b + \frac{ab}{100}$$

Applications:

- **Profit/Loss**: Based on cost price
- **Discount**: Reduction from marked price
- **Tax**: Addition to base amount
- **Interest**: Earning on principal

Ratio and Proportion:

Ratio: Comparison of two quantities

- a : b = a/b
- Properties: a:b = ka:kb (k ≠ 0)

Proportion: Equality of two ratios

- a : b :: c : d means a/b = c/d
- Cross multiplication: ad = bc

Types:

- **Direct Proportion**: As one increases, other increases

- **Inverse Proportion**: As one increases, other decreases

Applications:

- **Mixture problems**: Combining different ratios
- **Partnership**: Profit sharing based on investment ratios
- **Time and work**: Work rates in proportion to efficiency

Average and Weighted Average:

Simple Average:

$$\text{Average} = \frac{\text{Sum of all values}}{\text{Number of values}}$$

Properties:

- If all values increase by k, average increases by k
- If all values are multiplied by k, average is multiplied by k

Weighted Average:

$$\text{Weighted Average} = \frac{\sum(w_i \times x_i)}{\sum w_i}$$

where w_i are weights and x_i are values

Applications:

- **Academic grades**: Different subjects with different credits
- **Speed problems**: Average speed over different segments
- **Age problems**: Average age of groups

Algebra (25):

Linear Equations:

One Variable: ax + b = 0

- Solution: x = -b/a (if a ≠ 0)

Two Variables:

- ax + by = c
- dx + ey = f

Solution Methods:

1. **Substitution**: Solve one equation for one variable, substitute in other
2. **Elimination**: Add/subtract equations to eliminate one variable

3. **Cross multiplication**: For $a_1x + b_1y = c_1$ and $a_2x + b_2y = c_2$

$$x = \frac{b_1c_2 - b_2c_1}{a_1b_2 - a_2b_1}, \quad y = \frac{a_2c_1 - a_1c_2}{a_1b_2 - a_2b_1}$$

Quadratic Equations:

Standard Form: $ax^2 + bx + c = 0$ ($a \neq 0$)

Solution Methods:

1. **Factoring**: Express as $(px + q)(rx + s) = 0$
2. **Quadratic Formula**:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

3. **Completing the Square**: Convert to $(x + p)^2 = q$ form

Discriminant: $\Delta = b^2 - 4ac$

- $\Delta > 0$: Two distinct real roots
- $\Delta = 0$: One repeated real root
- $\Delta < 0$: No real roots (complex roots)

Relationship between Roots:
For roots α and β:

- Sum: $\alpha + \beta = -b/a$
- Product: $\alpha\beta = c/a$

Inequalities:

Linear Inequalities: $ax + b > 0$, $ax + b < 0$, etc.

- Solution is range of values
- Multiplication/division by negative number reverses inequality sign

Quadratic Inequalities: $ax^2 + bx + c > 0$, etc.

- Find roots of corresponding equation
- Test intervals between roots
- Consider sign of leading coefficient

Sequences and Series:

Arithmetic Progression (AP):

- General term: $a_n = a + (n-1)d$

- Sum of n terms: S_n = n/2[2a + (n-1)d] = n/2(first term + last term)

Geometric Progression (GP):

- General term: a_n = ar^(n-1)
- Sum of n terms: S_n = a(r^n - 1)/(r - 1) for r ≠ 1
- Sum to infinity: S∞ = a/(1-r) for |r| < 1

Special Series:

- Sum of first n natural numbers: 1 + 2 + ... + n = n(n+1)/2
- Sum of squares: $1^2 + 2^2 + ... + n^2 = n(n+1)(2n+1)/6$
- Sum of cubes: $1^3 + 2^3 + ... + n^3 = [n(n+1)/2]^2$

Logarithms:

Definition: If $a^x = b$, then x = log_a(b)

- a is base, b is argument, x is logarithm

Properties:

- log_a(xy) = log_a(x) + log_a(y)
- log_a(x/y) = log_a(x) - log_a(y)
- log_a(x^n) = n·log_a(x)
- log_a(a) = 1, log_a(1) = 0

Change of Base: log_a(x) = log_b(x)/log_b(a)

Common Logarithms:

- Natural logarithm: ln(x) = log_e(x)
- Common logarithm: log(x) = $\log_{10}(x)$

Geometry (20):

Basic Concepts:

Point: Has no dimension, represents location
Line: Extends infinitely in both directions
Ray: Has one endpoint, extends infinitely in one direction
Line Segment: Has two endpoints

Angles:

- **Acute**: < 90°
- **Right**: = 90°
- **Obtuse**: > 90° and < 180°

- **Straight**: = 180°
- **Reflex**: > 180° and < 360°
- **Complete**: = 360°

Angle Relationships:

- **Complementary**: Sum = 90°
- **Supplementary**: Sum = 180°
- **Vertically opposite**: Equal when two lines intersect
- **Corresponding**: Equal when parallel lines cut by transversal
- **Alternate interior**: Equal when parallel lines cut by transversal

Triangles:

Classification by Sides:

- **Equilateral**: All sides equal, all angles 60°
- **Isosceles**: Two sides equal, two angles equal
- **Scalene**: All sides different, all angles different

Classification by Angles:

- **Acute**: All angles < 90°
- **Right**: One angle = 90°
- **Obtuse**: One angle > 90°

Properties:

- Sum of angles = 180°
- Exterior angle = sum of two non-adjacent interior angles
- Sum of any two sides > third side (triangle inequality)

Area Formulas:

- General: Area = (1/2) × base × height
- Heron's formula: Area = $\sqrt{[s(s-a)(s-b)(s-c)]}$ where s = (a+b+c)/2
- Equilateral: Area = $(\sqrt{3}/4) \times \text{side}^2$

Special Triangles:

- **30-60-90**: Sides in ratio 1 : $\sqrt{3}$: 2
- **45-45-90**: Sides in ratio 1 : 1 : $\sqrt{2}$

Quadrilaterals:

Types:

- **Square**: All sides equal, all angles 90°
- **Rectangle**: Opposite sides equal, all angles 90°
- **Rhombus**: All sides equal, opposite angles equal
- **Parallelogram**: Opposite sides parallel and equal
- **Trapezium**: One pair of parallel sides

Properties:

- Sum of interior angles = 360°
- Diagonals of rectangle are equal
- Diagonals of rhombus bisect at right angles
- Diagonals of square are equal and bisect at right angles

Area Formulas:

- Square: side²
- Rectangle: length × width
- Parallelogram: base × height
- Rhombus: (1/2) × d_1 × d_2 (d_1, d_2 are diagonals)
- Trapezium: (1/2) × (sum of parallel sides) × height

Circles:

Basic Elements:

- **Radius**: Distance from center to circumference
- **Diameter**: Twice the radius, passes through center
- **Chord**: Line segment joining two points on circle
- **Arc**: Part of circumference
- **Sector**: Region bounded by two radii and arc
- **Segment**: Region bounded by chord and arc

Formulas:

- Circumference: $2\pi r$
- Area: πr^2
- Arc length: $(\theta/360°) \times 2\pi r$ (θ in degrees)
- Sector area: $(\theta/360°) \times \pi r^2$

Properties:

- Angle in semicircle = 90°
- Angles subtended by same arc are equal
- Tangent perpendicular to radius at point of contact
- Two tangents from external point are equal

Coordinate Geometry:

Distance Formula: Distance between (x_1, y_1) and (x_2, y_2)

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Section Formula: Point dividing line segment in ratio m:n

$$\left(\frac{mx_2 + nx_1}{m + n}, \frac{my_2 + ny_1}{m + n} \right)$$

Midpoint Formula:

$$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right)$$

Slope of Line:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Equation of Line:

- Point-slope form: $y - y_1 = m(x - x_1)$
- Slope-intercept form: $y = mx + c$
- Two-point form: $(y - y_1)/(y_2 - y_1) = (x - x_1)/(x_2 - x_1)$

Mensuration (15):

2D Figures:

Triangle: Area = (1/2) × base × height
Rectangle: Area = length × width, Perimeter = 2(length + width)
Square: Area = side², Perimeter = 4 × side
Circle: Area = πr², Circumference = 2πr
Parallelogram: Area = base × height
Rhombus: Area = (1/2) × d_1 × d_2
Trapezium: Area = (1/2) × (a + b) × h

3D Figures:

Cube:

- Volume = side³
- Surface area = 6 × side²

Cuboid (Rectangular Prism):

- Volume = length × width × height
- Surface area = 2(lw + wh + lh)

Cylinder:

- Volume = $\pi r^2 h$
- Curved surface area = $2\pi rh$
- Total surface area = $2\pi r(r + h)$

Cone:

- Volume = $(1/3)\pi r^2 h$
- Curved surface area = πrl (l = slant height)
- Total surface area = $\pi r(r + l)$

Sphere:

- Volume = $(4/3)\pi r^3$
- Surface area = $4\pi r^2$

Hemisphere:

- Volume = $(2/3)\pi r^3$
- Curved surface area = $2\pi r^2$
- Total surface area = $3\pi r^2$

Data Interpretation (10):

Types of Data Representation:

Tables: Data arranged in rows and columns

- Read row and column headers carefully
- Look for totals, subtotals, percentages
- Calculate missing values using given information

Bar Charts:

- **Vertical**: Categories on x-axis, values on y-axis
- **Horizontal**: Categories on y-axis, values on x-axis
- **Grouped**: Multiple bars for each category
- **Stacked**: Parts of whole shown in single bar

Line Graphs: Show trends over time

- Identify increasing/decreasing trends
- Find maximum/minimum points
- Calculate rate of change between points

Pie Charts: Show parts of whole

- Each sector represents percentage of total
- Central angle = (value/total) × 360°
- Compare sectors by size

Histograms: Show frequency distribution

- Bars touch each other (continuous data)
- Width represents class interval
- Height represents frequency

Common Calculations:

Percentage: (Part/Whole) × 100
Percentage Change: ((New - Old)/Old) × 100
Average: Sum of values / Number of values
Ratio: Comparison between quantities
Growth Rate: ((Final - Initial)/Initial) × 100

Problem-Solving Tips:

For Arithmetic:

1. Learn divisibility rules and multiplication tables
2. Practice mental calculation techniques
3. Use approximation for complex calculations
4. Remember common fraction-decimal-percentage conversions
5. Break complex problems into simpler steps

For Algebra:

1. Identify the type of equation/inequality
2. Use appropriate solution method
3. Check solutions by substitution
4. Be careful with signs when manipulating inequalities
5. Practice word problems to improve translation skills

For Geometry:

1. Draw diagrams for visualization
2. Identify given information and what to find
3. Use appropriate formulas and theorems
4. Check if answer makes geometric sense
5. Remember special triangle ratios

For Data Interpretation:

1. Read titles, labels, and legends carefully
2. Understand scale and units
3. Look for patterns and trends
4. Use approximation for quick calculations
5. Double-check calculations

GATE Tips:

- Time management crucial: allocate time based on marks
- Use elimination method for multiple choice questions
- Don't spend too much time on any single question
- Practice mental math for speed
- Learn shortcuts and tricks for common calculations
- Review basic formulas regularly
- Solve previous years' questions for pattern recognition
- Focus on accuracy over speed initially, then build speed

Common Shortcuts:

Multiplication:

- $(a + b)(a - b) = a^2 - b^2$
- $(a + b)^2 = a^2 + 2ab + b^2$
- $(a - b)^2 = a^2 - 2ab + b^2$

Percentage:

- 10% of x = x/10
- 25% of x = x/4
- 50% of x = x/2

Squares:

- $(50 + a)^2 = 2500 + 100a + a^2$
- Numbers ending in 5: $25^2 = 625$, $35^2 = 1225$

Division:

- Divisibility by 4: Last two digits divisible by 4
- Divisibility by 8: Last three digits divisible by 8
- Divisibility by 25: Last two digits are 00, 25, 50, or 75

Volume 2: Core Computer Science Subjects

Volume 2: Core Computer Science Subjects

1. Algorithms (334)

1.1 Algorithm Design Paradigms (8)

Key Concepts: Systematic approach to problem-solving using algorithmic paradigms.

Greedy Algorithms

Definition: Makes locally optimal choices at each step with the hope of finding a global optimum.

Characteristics:

- **Greedy Choice Property**: A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
- **Optimal Substructure**: An optimal solution to the problem contains optimal solutions to subproblems

Algorithm Framework:

1. Make a greedy choice
2. Reduce the problem to a smaller subproblem
3. Solve the subproblem recursively/iteratively

Common Greedy Algorithms:

- **Dijkstra's Algorithm**: Single-source shortest paths
- **Prim's & Kruskal's**: Minimum Spanning Tree
- **Huffman Coding**: Data compression
- **Activity Selection**: Maximum number of non-overlapping activities
- **Fractional Knapsack**: Maximize value with weight constraint

Activity Selection Problem:

```
def activity_selection(activities):
    # Sort by finish time
    activities.sort(key=lambda x: x[1])
    selected = [activities[0]]
    last_finish = activities[0][1]

    for i in range(1, len(activities)):
        if activities[i][0] >= last_finish:  # Start time >= last finish
            selected.append(activities[i])
```

```
            last_finish = activities[i][1]

    return selected
```

Fractional Knapsack:

```
def fractional_knapsack(items, capacity):
    # Items: (value, weight)
    items.sort(key=lambda x: x[0]/x[1], reverse=True)  # Sort by
value/weight ratio
    total_value = 0
    current_weight = 0

    for value, weight in items:
        if current_weight + weight <= capacity:
            total_value += value
            current_weight += weight
        else:
            remaining = capacity - current_weight
            fraction = remaining / weight
            total_value += value * fraction
            break

    return total_value
```

When Greedy Fails:

- **0/1 Knapsack**: Cannot take fractions, greedy by value/weight ratio fails
- **Traveling Salesman Problem**: Greedy nearest neighbor doesn't guarantee optimal solution
- **Longest Path Problem**: Greedy choices can lead to suboptimal paths

Proof Techniques:

1. **Exchange Argument**: Show that any optimal solution can be transformed into greedy solution without decreasing quality
2. **Stay Ahead**: Show that greedy solution is at least as good as optimal solution at every step
3. **Cut and Paste**: If greedy choice leads to suboptimal solution, replace part with better choice

GATE Tips:

- Always verify greedy choice property and optimal substructure
- Counterexamples are powerful tools to disprove greedy approaches
- Time complexity often dominated by sorting step: O(n log n)

- Space complexity typically O(1) or O(n) for storing results

Dynamic Programming

Definition: Solves problems by breaking them down into overlapping subproblems and storing solutions to avoid recomputation.

Characteristics:

- **Optimal Substructure**: Optimal solution contains optimal solutions to subproblems
- **Overlapping Subproblems**: Subproblems recur multiple times

Algorithm Framework:

1. Characterize structure of optimal solution
2. Define value of optimal solution recursively
3. Compute value bottom-up (tabulation) or top-down with memoization
4. Construct optimal solution from computed information

Approaches:

- **Top-Down (Memoization)**: Recursive with caching
- **Bottom-Up (Tabulation)**: Iterative, fill table systematically

Common DP Problems:

- **Matrix Chain Multiplication**: Minimize scalar multiplications
- **Longest Common Subsequence (LCS)**: Find longest common subsequence
- **0/1 Knapsack**: Maximize value with weight constraint
- **All Pairs Shortest Paths (Floyd-Warshall)**: Shortest paths between all pairs
- **Optimal Binary Search Tree**: Minimize expected search cost

Matrix Chain Multiplication:

```
def matrix_chain_order(p):
    n = len(p) - 1  # Number of matrices
    m = [[0] * (n + 1) for _ in range(n + 1)]
    s = [[0] * (n + 1) for _ in range(n + 1)]

    for l in range(2, n + 1):  # l = chain length
        for i in range(1, n - l + 2):
            j = i + l - 1
            m[i][j] = float('inf')
            for k in range(i, j):
                cost = m[i][k] + m[k + 1][j] + p[i-1] * p[k] * p[j]
                if cost < m[i][j]:
                    m[i][j] = cost
```

```
                    s[i][j] = k

    return m, s
```

Longest Common Subsequence:

```
def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    # Reconstruct LCS
    lcs_str = []
    i, j = m, n
    while i > 0 and j > 0:
        if X[i-1] == Y[j-1]:
            lcs_str.append(X[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1

    return ''.join(reversed(lcs_str)), dp[m][n]
```

Space Optimization Techniques:

- **Rolling Array**: Use only previous row/column instead of entire table
- **State Compression**: Reduce dimensionality using bit masks
- **Divide and Conquer DP**: For specific recurrence structures

GATE Tips:

- Identify if problem has optimal substructure and overlapping subproblems
- Define state clearly: what does dp[i][j] represent?
- Transition: How to compute current state from previous states?
- Base cases: Initialize properly
- Time complexity: O(states × transitions)
- Space complexity: Can often be optimized from $O(n^2)$ to $O(n)$

Comparison: Greedy vs DP

Aspect	Greedy	Dynamic Programming
Choice	Makes irrevocable choice at each step	Considers all choices, picks optimal
Subproblems	No overlapping subproblems	Has overlapping subproblems
Optimality	Not always optimal	Always optimal (when applicable)
Complexity	Usually faster: O(n log n)	Slower: $O(n^2)$ to $O(n^3)$
Memory	O(1) or O(n)	$O(n^2)$ or more
Examples	Dijkstra, MST	Knapsack, LCS, Matrix Chain

1.2 Algorithm Design Techniques (9)

Divide and Conquer

Definition: Breaks problem into independent subproblems of same type, solves them recursively, and combines solutions.

Algorithm Framework:

1. **Divide**: Partition problem into smaller subproblems
2. **Conquer**: Solve subproblems recursively (base case if small enough)
3. **Combine**: Merge solutions of subproblems into final solution

Recurrence Relations:

- General form: T(n) = aT(n/b) + f(n)
- a = number of subproblems
- n/b = size of each subproblem
- f(n) = cost of divide and combine steps

Common Algorithms:

- **Merge Sort**: T(n) = 2T(n/2) + O(n) = O(n log n)
- **Quick Sort**: T(n) = T(k) + T(n-k-1) + O(n), average O(n log n), worst $O(n^2)$
- **Binary Search**: T(n) = T(n/2) + O(1) = O(log n)
- **Strassen's Matrix Multiplication**: T(n) = 7T(n/2) + $O(n^2)$ = O(n^2.81)
- **Closest Pair of Points**: T(n) = 2T(n/2) + O(n log n) = O(n log n)

Merge Sort Implementation:

```
def merge_sort(arr):
    if len(arr) <= 1:
```

```
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Quick Sort Implementation:

```
def quick_sort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1

    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high]  # Choose last element as pivot
    i = low - 1  # Index of smaller element

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

Master Theorem Applications:

Case	**Condition**	**Solution**	**Examples**
Case 1	f(n) = O(n^log_b(a-ε))	T(n) = Θ(n^log_b(a))	Merge Sort: f(n)=O(n), $\log_2 2=1$
Case 2	f(n) = Θ(n^log_b(a))	T(n) = Θ(n^log_b(a) log n)	Binary Search: f(n)=O(1), $\log_2 1=0$
Case 3	f(n) = Ω(n^log_b(a+ε)) and af(n/b) ≤ cf(n)	T(n) = Θ(f(n))	Strassen: f(n)=O(n²), $\log_2 7\approx 2.81$

Backtracking

Definition: Systematic way to iterate through all possible configurations of a problem, abandoning ("backtracking") partial candidates that cannot lead to valid solutions.

Algorithm Framework:

1. Start with empty solution
2. Add next candidate to partial solution
3. Check if partial solution can lead to valid solution
4. If yes, continue recursively
5. If no, backtrack and try next candidate
6. If all candidates exhausted, backtrack further

Key Components:

- **Solution Vector**: x[1..n] represents current partial solution
- **Constraint Function**: Checks if partial solution is promising
- **Objective Function**: Evaluates quality of complete solution
- **Bounding Function**: Prunes unpromising branches

Common Algorithms:

- **N-Queens Problem**: Place N queens on NxN chessboard
- **Graph Coloring**: Color vertices with minimum colors
- **Subset Sum**: Find subset with given sum
- **Hamiltonian Cycle**: Find cycle visiting each vertex exactly once
- **Sudoku Solver**: Fill Sudoku grid following rules

N-Queens Implementation:

```
def solve_n_queens(n):
    solutions = []
    board = [-1] * n  # board[i] = column of queen in row i

    def is_safe(row, col):
        for r in range(row):
```

```
            c = board[r]
            if c == col or abs(r - row) == abs(c - col):
                return False
        return True

    def backtrack(row):
        if row == n:
            solutions.append(board.copy())
            return

        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                backtrack(row + 1)
                board[row] = -1  # Backtrack

    backtrack(0)
    return solutions
```

Branch and Bound Optimization:

- **Lower Bound**: Minimum possible cost from current state
- **Upper Bound**: Best solution found so far
- **Pruning**: Skip branches where lower bound ≥ upper bound

GATE Tips:

- Time complexity often exponential: O(b^d) where b=branching factor, d=depth
- Space complexity: O(n) for recursion stack
- Backtracking is complete (finds all solutions) but not efficient for large n
- Branch and bound improves efficiency but still exponential in worst case
- Look for symmetry and pruning opportunities to reduce search space

Transform and Conquer

Definition: Transforms problem into a different representation that is easier to solve.

Techniques:

- **Instance Simplification**: Reduce to simpler instance of same problem
- **Representation Change**: Change data representation
- **Problem Reduction**: Reduce to different problem

Examples:

- **Presorting**: Sort array before processing (e.g., finding duplicates)
- **AVL Trees**: Self-balancing BSTs for efficient operations

- **Fast Fourier Transform (FFT)**: Converts convolution to multiplication
- **Reduction to Max Flow**: Many problems can be reduced to max flow

1.3-1.4 Asymptotic Notation (21)

Formal Definitions

Big-O Notation (Upper Bound):

- **Definition**: f(n) = O(g(n)) if ∃ positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- **Intuition**: f(n) grows no faster than g(n) asymptotically
- **Example**: $3n^2 + 2n + 1 = O(n^2)$

Big-Ω Notation (Lower Bound):

- **Definition**: f(n) = Ω(g(n)) if ∃ positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$
- **Intuition**: f(n) grows at least as fast as g(n) asymptotically
- **Example**: $3n^2 + 2n + 1 = \Omega(n^2)$

Big-Θ Notation (Tight Bound):

- **Definition**: f(n) = Θ(g(n)) if ∃ positive constants c_1, c_2 and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$
- **Intuition**: f(n) grows exactly like g(n) asymptotically
- **Example**: $3n^2 + 2n + 1 = \Theta(n^2)$

Little-o Notation (Strict Upper Bound):

- **Definition**: f(n) = o(g(n)) if ∀ positive constants $c > 0$, ∃ n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$
- **Intuition**: f(n) grows strictly slower than g(n)
- **Example**: $n = o(n^2)$, but $n^2 \neq o(n^2)$

Little-ω Notation (Strict Lower Bound):

- **Definition**: f(n) = ω(g(n)) if ∀ positive constants $c > 0$, ∃ n_0 such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$
- **Intuition**: f(n) grows strictly faster than g(n)
- **Example**: $n^2 = \omega(n)$, but $n^2 \neq \omega(n^2)$

Properties and Theorems

Transitivity:

- If f = O(g) and g = O(h), then f = O(h)

- If f = Ω(g) and g = Ω(h), then f = Ω(h)
- If f = Θ(g) and g = Θ(h), then f = Θ(h)

Reflexivity:

- f = O(f), f = Ω(f), f = Θ(f)

Symmetry:

- f = Θ(g) iff g = Θ(f)
- f = O(g) iff g = Ω(f)

Common Function Growth Rates:

Order	**Function**	**Example Algorithm**
O(1)	Constant	Array access
O(log log n)	Double logarithmic	Van Emde Boas tree
O(log n)	Logarithmic	Binary search
O(√n)	Square root	Trial division primality
O(n)	Linear	Linear search
O(n log n)	Linearithmic	Merge sort, Heap sort
O(n^2)	Quadratic	Bubble sort, Insertion sort
O(n^3)	Cubic	Matrix multiplication (naive)
O(2^n)	Exponential	Subset generation
O(n!)	Factorial	Permutation generation

Master Theorem (Detailed)

Standard Form: T(n) = aT(n/b) + f(n), where a ≥ 1, b > 1

Case 1: If f(n) = O(n^log_b(a-ε)) for some ε > 0

- **Solution**: T(n) = Θ(n^log_b(a))
- **Example**: T(n) = 2T(n/2) + O(1) → a=2, b=2, $\log_2 2=1$, f(n)=O(1)=O(n^0)
  Since 0 < 1, T(n) = Θ(n)

Case 2: If f(n) = Θ(n^log_b(a) log^k n) for k ≥ 0

- **Solution**: T(n) = Θ(n^log_b(a) log^(k+1) n)
- **Example**: T(n) = 2T(n/2) + O(n) → a=2, b=2, $\log_2 2=1$, f(n)=O(n)=Θ(n^1)
  T(n) = Θ(n log n)

Case 3: If f(n) = Ω(n^log_b(a+ε)) for some ε > 0, and af(n/b) ≤ cf(n) for some c < 1 and large n

- **Solution**: T(n) = Θ(f(n))

- **Example**: T(n) = T(n/2) + O(n) → a=1, b=2, $\log_2 1=0$, f(n)=O(n)=Ω(n^0+ε)
  T(n) = Θ(n)

Extensions:

- **Akra-Bazzi Method**: For recurrences like T(n) = Σa_iT(b_in) + f(n)
- **Recursion Tree Method**: Visualize and sum costs at each level
- **Substitution Method**: Guess solution and prove by induction

Practical Analysis Techniques

Amortized Analysis:

- **Aggregate Method**: Total cost of n operations / n
- **Accounting Method**: Assign amortized cost to operations
- **Potential Method**: Define potential function Φ, amortized cost = actual cost + ΔΦ

Example - Dynamic Array:

- Insert operation: O(1) amortized
- When array full, double size: O(n) operation
- Amortized cost: 3 per insertion (2 for insertion, 1 stored for future resizing)

Probabilistic Analysis:

- **Expected Running Time**: Average over all possible inputs
- **Randomized Algorithms**: Use randomness to achieve good expected performance

GATE Tips:

- Always specify which case of Master Theorem you're using
- For Θ notation, both O and Ω bounds must hold
- When comparing functions, use limits: lim(n→∞) f(n)/g(n)
- Common mistakes: confusing O with Θ, ignoring lower-order terms
- For recursive algorithms, write recurrence first, then solve
- For iterative algorithms, count loop iterations and operations per iteration

1.5 Bellman-Ford Algorithm (2)

Key Concepts: Single-source shortest paths algorithm that handles negative edge weights and detects negative cycles.

Problem Statement: Given weighted directed graph G = (V, E) with weight function w: E → $\mathbb{R}$, and source vertex s, find shortest paths from s to all other vertices.

Algorithm:

```python
def bellman_ford(graph, source):
    # Initialize distances
    dist = {v: float('inf') for v in graph.vertices}
    dist[source] = 0
    predecessor = {v: None for v in graph.vertices}

    # Relax all edges |V|-1 times
    for i in range(len(graph.vertices) - 1):
        for u, v, weight in graph.edges:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                predecessor[v] = u

    # Check for negative cycles
    for u, v, weight in graph.edges:
        if dist[u] + weight < dist[v]:
            raise ValueError("Graph contains negative cycle")

    return dist, predecessor
```

Correctness Proof:

- **Lemma 1**: After k iterations, dist[v] is at most the weight of any path from s to v with at most k edges
- **Lemma 2**: If no negative cycles reachable from s, then after |V|-1 iterations, dist[v] = δ(s,v) for all v
- **Lemma 3**: If there exists a negative cycle reachable from s, the algorithm detects it

Time Complexity: O(VE)

- |V|-1 iterations
- Each iteration relaxes all |E| edges
- Negative cycle check: O(E)

Space Complexity: O(V)

- Distance array: O(V)
- Predecessor array: O(V)

Optimizations:

- **Early Termination**: Stop if no updates in an iteration
- **Queue-based (SPFA)**: Only relax edges from vertices whose distance changed
  - Average case: O(E)
  - Worst case: O(VE)

Applications:

- **Distance Vector Routing**: Network routing protocols
- **Currency Arbitrage**: Detect profitable currency exchange cycles
- **Difference Constraints**: Solve systems of inequalities
- **Job Scheduling**: Find feasible schedule with precedence constraints

Comparison with Dijkstra:
Aspect	**Bellman-Ford**	**Dijkstra**
Negative Weights	Handles negative weights	Fails with negative weights
Negative Cycles	Detects negative cycles	Cannot detect
Time Complexity	O(VE)	O((V+E)log V) with heap
Data Structure	Simple arrays	Priority queue
Use Case	General graphs	Non-negative weights

GATE Tips:

- Bellman-Ford can handle negative weights but not negative cycles (detects them)
- Requires |V|-1 iterations for correctness
- One extra iteration needed to detect negative cycles
- Space complexity is O(V), better than Floyd-Warshall's $O(V^2)$
- For sparse graphs, better than Floyd-Warshall; for dense graphs, worse
- Can be parallelized easily across edges

1.6 Binary Search (Extended Comprehensive Coverage)

Core Concepts and Variations

Fundamental Principle: Works on sorted arrays or monotonic functions by repeatedly dividing search space in half.

Mathematical Foundation:

- **Monotonicity**: Function f is monotonic if $f(i) \leq f(j)$ for all $i < j$ (non-decreasing) or $f(i) \geq f(j)$ (non-increasing)
- **Search Space**: [low, high] where solution must lie
- **Invariant**: At each step, if solution exists, it lies within current search space

Termination Conditions:

- **Standard (Exact Match)**: `while (low <= high)`
- **Lower/Upper Bound**: `while (low < high)`
- **Floating Point**: `while (high - low > epsilon)`

Advanced Variations and Applications

1. Binary Search on Real Numbers

```python
def sqrt(x, epsilon=1e-7):
    if x < 0:
        raise ValueError("Cannot compute square root of negative number")
    if x == 0:
        return 0

    low, high = 0, max(1, x)

    while high - low > epsilon:
        mid = (low + high) / 2
        if mid * mid <= x:
            low = mid
        else:
            high = mid

    return (low + high) / 2
```

2. Binary Search on Unimodal Functions

```python
def find_max_unimodal(f, low, high, epsilon=1e-7):
    while high - low > epsilon:
        mid1 = low + (high - low) / 3
        mid2 = high - (high - low) / 3

        if f(mid1) < f(mid2):
            low = mid1
        else:
            high = mid2

    return (low + high) / 2
```

3. Binary Search with Predicate Functions

```python
def binary_search_predicate(predicate, low, high):
    """
    Find smallest x such that predicate(x) is True
    Predicate must be of form: F, F, ..., F, T, T, ..., T
    """
    while low < high:
        mid = low + (high - low) // 2
        if predicate(mid):
            high = mid
        else:
            low = mid + 1

    return low if predicate(low) else -1
```

4. Binary Search in Rotated Sorted Arrays with Duplicates

```python
def search_rotated_with_duplicates(nums, target):
    low, high = 0, len(nums) - 1

    while low <= high:
        mid = low + (high - low) // 2

        if nums[mid] == target:
            return True

        # Handle duplicates
        if nums[low] == nums[mid] == nums[high]:
            low += 1
            high -= 1
        # Left half is sorted
        elif nums[low] <= nums[mid]:
            if nums[low] <= target < nums[mid]:
                high = mid - 1
            else:
                low = mid + 1
        # Right half is sorted
        else:
            if nums[mid] < target <= nums[high]:
                low = mid + 1
            else:
                high = mid - 1

    return False
```

5. Binary Search for K-th Smallest Element in Sorted Matrix

```python
def kth_smallest(matrix, k):
    n = len(matrix)
    low, high = matrix[0][0], matrix[n-1][n-1]

    while low < high:
        mid = low + (high - low) // 2
        count = 0
        j = n - 1

        # Count elements <= mid
        for i in range(n):
            while j >= 0 and matrix[i][j] > mid:
                j -= 1
            count += j + 1

        if count < k:
```

```
            low = mid + 1
        else:
            high = mid

    return low
```

Mathematical Analysis and Proofs

Loop Invariant Proof:
For standard binary search:

- **Invariant**: At start of each iteration, if target exists in array, it must be in range [low, high]
- **Initialization**: Initially low=0, high=n-1, so target (if exists) is in [0, n-1]
- **Maintenance**: Each iteration maintains invariant by eliminating half that cannot contain target
- **Termination**: Loop terminates when low > high (target not found) or when arr[mid] == target

Time Complexity Analysis:

- **Recurrence**: T(n) = T(n/2) + O(1)
- **Master Theorem**: a=1, b=2, f(n)=O(1)
  - log_b(a) = $\log_2(1) = 0$
  - f(n) = O(n^0), so Case 2 applies
  - T(n) = Θ(log n)
- **Intuitive**: Each iteration halves search space, so after k iterations, space size = n/2^k
  - When n/2^k = 1, k = $\log_2(n)$

Space Complexity:

- **Iterative**: O(1) - only maintains low, high, mid pointers
- **Recursive**: O(log n) - recursion depth equals number of iterations

GATE-Specific Problem Patterns

Pattern 1: Minimize Maximum/Maximize Minimum

```
def minimize_max_allocation(pages, students):
    def can_allocate(max_pages):
        current_sum = 0
        student_count = 1

        for pages_i in pages:
            if pages_i > max_pages:
                return False
            if current_sum + pages_i > max_pages:
```

```
                student_count += 1
                current_sum = pages_i
                if student_count > students:
                    return False
            else:
                current_sum += pages_i

        return True

    low, high = max(pages), sum(pages)
    result = high

    while low <= high:
        mid = (low + high) // 2
        if can_allocate(mid):
            result = mid
            high = mid - 1
        else:
            low = mid + 1

    return result
```

Pattern 2: Find Transition Point

```
def find_transition_point(arr):
    # Find index where 0s end and 1s start
    low, high = 0, len(arr) - 1

    while low < high:
        mid = low + (high - low) // 2
        if arr[mid] == 0:
            low = mid + 1
        else:
            high = mid

    return low if arr[low] == 1 else -1
```

Pattern 3: Binary Search on Answer with Constraints

```
def find_min_capacity(weights, days):
    def can_ship(capacity):
        current_load = 0
        day_count = 1

        for weight in weights:
            if current_load + weight > capacity:
                day_count += 1
                current_load = weight
```

```
                if day_count > days:
                    return False
            else:
                current_load += weight

        return True

    low, high = max(weights), sum(weights)

    while low < high:
        mid = (low + high) // 2
        if can_ship(mid):
            high = mid
        else:
            low = mid + 1

    return low
```

Common Pitfalls and Solutions

Pitfall 1: Integer Overflow in Mid Calculation

- **Wrong**: `mid = (low + high) // 2`
- **Correct**: `mid = low + (high - low) // 2`
- **Alternative**: `mid = (low + high) >>> 1` (bitwise right shift)

Pitfall 2: Infinite Loops

- **Cause**: Incorrect update of low/high bounds
- **Solution**: Ensure progress in each iteration
- **Pattern**:
  - For `while (low <= high)`: always update low = mid + 1 or high = mid - 1
  - For `while (low < high)`: use low = mid + 1 or high = mid

Pitfall 3: Off-by-One Errors

- **Test Cases**: Empty array, single element, target at boundaries
- **Solution**: Write test cases for edge conditions:
  - n = 0, n = 1
  - target < all elements, target > all elements
  - target at first/last position
  - duplicate elements

Pitfall 4: Incorrect Termination Condition

- **Lower Bound**: Needs `while (low < high)` to avoid infinite loops
- **Exact Match**: Needs `while (low <= high)` to check middle element

- **Solution**: Match termination condition to problem requirements

Advanced Applications in Computer Science

1. Database Indexing (B+ Trees)

- Binary search principles applied to disk-based data structures
- Each node contains multiple keys, binary search within node
- Time complexity: O(log_B N) where B = branching factor

2. Memory Management

- **Buddy System**: Binary search for free blocks of appropriate size
- **Slab Allocator**: Binary search for object size classes
- **Virtual Memory**: Binary search in page tables

3. Computational Geometry

- **Point Location**: Binary search in planar subdivisions
- **Range Queries**: k-d trees use binary search principles
- **Convex Hull**: Binary search for tangent points

4. Machine Learning

- **Hyperparameter Tuning**: Binary search for optimal learning rate
- **Cross-Validation**: Binary search for optimal model complexity
- **Feature Selection**: Binary search for minimal feature set

5. Network Protocols

- **TCP Congestion Control**: Binary search for optimal window size
- **Routing**: Binary search in routing tables
- **Load Balancing**: Binary search for optimal server assignment

GATE Exam Strategy

Time Complexity Questions:

- Always O(log n) for standard binary search
- For modified versions, analyze carefully:
  - Binary search on matrix: O(n log n) vs O(log n^2) = O(log n)
  - With duplicates: Worst case O(n) when all elements same

Space Complexity Questions:

- Iterative: O(1)
- Recursive: O(log n) due to call stack

- With additional data structures: O(n) or more

Problem Identification:

1. **Is the array sorted?** If yes, binary search likely applicable
2. **Is there a monotonic property?** If function is monotonic, binary search on answer
3. **Can search space be divided?** If yes/no question can eliminate half, binary search applicable
4. **Are we optimizing something?** Minimize maximum or maximize minimum often uses binary search

Common GATE Question Types:

- Time complexity of given binary search implementation
- Correct implementation of binary search variant
- Finding bugs in binary search code
- Application to specific problems (allocation, capacity, etc.)
- Comparison with other search algorithms

1.7 Bitonic Array Search (1)

Extended Coverage:

Definition and Properties

Bitonic Sequence: A sequence that first increases monotonically and then decreases monotonically, or vice versa.

Mathematical Definition: Array A[0..n-1] is bitonic if there exists an index k ($0 \leq k < n$) such that:

- $A[0] \leq A[1] \leq ... \leq A[k] \geq A[k+1] \geq ... \geq A[n-1]$ (increasing-then-decreasing)
- OR $A[0] \geq A[1] \geq ... \geq A[k] \leq A[k+1] \leq ... \leq A[n-1]$ (decreasing-then-increasing)

Special Cases:

- **Entirely Increasing**: k = n-1 (peak at end)
- **Entirely Decreasing**: k = 0 (peak at start)
- **Rotated Bitonic**: Circular shift of bitonic sequence
- **Strict Bitonic**: No equal adjacent elements

Properties:

- **Single Peak/Valley**: Exactly one local maximum/minimum
- **Unimodal**: Function increases to peak, then decreases
- **Searchable in O(log n)**: Can find peak and search both halves

Algorithms and Implementations

1. Finding Peak Element in Bitonic Array

```
def find_peak_bitonic(arr):
    low, high = 0, len(arr) - 1

    while low < high:
        mid = low + (high - low) // 2

        # If mid element is greater than next element
        if arr[mid] > arr[mid + 1]:
            high = mid  # Peak is in left half including mid
        else:
            low = mid + 1  # Peak is in right half

    return low  # low == high is the peak index
```

2. Searching in Bitonic Array

```
def bitonic_search(arr, target):
    n = len(arr)
    if n == 0:
        return -1

    # Find peak element
    peak = find_peak_bitonic(arr)

    # Search in increasing part [0...peak]
    result = binary_search(arr, 0, peak, target)
    if result != -1:
        return result

    # Search in decreasing part [peak+1...n-1]
    return binary_search_descending(arr, peak + 1, n - 1, target)

def binary_search_descending(arr, low, high, target):
    while low <= high:
        mid = low + (high - low) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] > target:
            low = mid + 1  # Search right in decreasing array
        else:
            high = mid - 1  # Search left in decreasing array

    return -1
```

3. Searching in Rotated Bitonic Array

```
def rotated_bitonic_search(arr, target):
    n = len(arr)
    if n == 0:
        return -1

    # Find the rotation point (minimum element)
    min_idx = find_min_rotated(arr)

    # Determine which half is bitonic
    if arr[0] <= arr[min_idx - 1]:  # Left half is bitonic
        if min_idx > 0 and arr[0] <= target <= arr[min_idx - 1]:
            return bitonic_search(arr[:min_idx], target)

    # Right half must be bitonic
    if min_idx < n and arr[min_idx] <= target <= arr[-1]:
        result = bitonic_search(arr[min_idx:], target)
        return min_idx + result if result != -1 else -1

    return -1

def find_min_rotated(arr):
    low, high = 0, len(arr) - 1

    while low < high:
        mid = low + (high - low) // 2

        if arr[mid] > arr[high]:
            low = mid + 1
        else:
            high = mid

    return low
```

Mathematical Analysis

Time Complexity:

- **Finding Peak**: O(log n) - each step eliminates half the array
- **Searching**: O(log n) for each half, total O(log n)
- **Rotated Version**: O(log n) for finding rotation + O(log n) for search = O(log n)

Space Complexity: O(1) for iterative implementations

Correctness Proof:

- **Peak Finding**:

    - Base case: n=1, peak is the only element
    - Inductive step: If arr[mid] > arr[mid+1], peak is in left half; else in right half
    - Termination: low == high when peak found
- **Search Correctness**:
    - Increasing part: Standard binary search works
    - Decreasing part: Modified binary search with reversed comparisons
    - One of the parts must contain target if it exists

Applications

1. Signal Processing:

- **Peak Detection**: Finding maximum signal strength
- **Edge Detection**: Finding transitions in signals
- **Pattern Recognition**: Identifying characteristic shapes

2. Optimization Problems:

- **Unimodal Function Optimization**: Finding maximum/minimum
- **Resource Allocation**: Optimal distribution with single peak efficiency
- **Economic Models**: Finding optimal price/output levels

3. Computer Graphics:

- **Illumination Models**: Finding maximum brightness
- **Animation Curves**: Smooth transitions with single peak acceleration
- **Terrain Generation**: Creating natural-looking landscapes

4. Scientific Computing:

- **Molecular Dynamics**: Finding energy minima
- **Quantum Mechanics**: Finding wavefunction peaks
- **Statistical Analysis**: Finding mode in unimodal distributions

GATE-Specific Insights

Common Question Types:

1. **Time Complexity**: Finding peak in bitonic array - O(log n)
2. **Algorithm Selection**: Which algorithm to use for bitonic search
3. **Edge Cases**: Handling entirely increasing/decreasing arrays
4. **Modified Problems**: Searching in rotated bitonic arrays
5. **Proof Questions**: Proving correctness of peak finding algorithm

Tricks and Shortcuts:

- **Single Pass Search**: Can sometimes search without finding peak first
- **Modified Binary Search**: Use property that one side must be sorted
- **Peak as Pivot**: Use peak to divide array into two sorted subarrays

Problem-Solving Strategy:

1. **Identify Bitonic Property**: Check if array has single peak/valley
2. **Find Critical Point**: Locate peak/valley first
3. **Divide and Conquer**: Split into monotonic segments
4. **Apply Standard Search**: Use binary search on appropriate segment
5. **Handle Rotation**: If rotated, find rotation point first

Sample GATE Questions:

1. What is time complexity of finding maximum element in bitonic array?
   - Answer: O(log n)
2. How to search for element in bitonic array?
   - Answer: Find peak, then binary search in increasing and decreasing parts
3. What is the minimum number of comparisons needed to find peak in bitonic array of size n?
   - Answer: $\lceil \log_2 n \rceil$ in worst case

1.8 Depth First Search (DFS) (Extended)

Key Concepts: Graph traversal algorithm that explores as far as possible along each branch before backtracking. Uses stack (explicit or implicit via recursion).

Algorithm and Implementation

Recursive DFS:

```
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=' ')  # Process node

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

    return visited
```

Iterative DFS:

```
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()

        if node not in visited:
            visited.add(node)
            print(node, end=' ')  # Process node

            # Add neighbors in reverse order to maintain same order as
recursive
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)

    return visited
```

DFS with Parent Tracking:

```
def dfs_with_parent(graph, start):
    visited = {start: None}  # node: parent
    stack = [start]

    while stack:
        node = stack.pop()

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited[neighbor] = node
                stack.append(neighbor)

    return visited
```

Advanced Applications and Variations

1. Cycle Detection in Directed Graphs

```
def has_cycle_directed(graph):
    visited = set()
    recursion_stack = set()

    def dfs(node):
        visited.add(node)
        recursion_stack.add(node)
```

```
            for neighbor in graph[node]:
                if neighbor not in visited:
                    if dfs(neighbor):
                        return True
                elif neighbor in recursion_stack:
                    return True

            recursion_stack.remove(node)
            return False

        for node in graph:
            if node not in visited:
                if dfs(node):
                    return True

        return False
```

2. Topological Sort (DFS-based)

```
def topological_sort_dfs(graph):
    visited = set()
    result = []

    def dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)
        result.append(node)  # Add after processing all descendants

    for node in graph:
        if node not in visited:
            dfs(node)

    return result[::-1]  # Reverse to get topological order
```

3. Strongly Connected Components (Kosaraju's Algorithm)

```
def kosaraju_scc(graph):
    # Step 1: First DFS to get finishing times
    visited = set()
    order = []

    def dfs1(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs1(neighbor)
```

```
        order.append(node)

    for node in graph:
        if node not in visited:
            dfs1(node)

    # Step 2: Transpose graph
    transpose = {node: [] for node in graph}
    for u in graph:
        for v in graph[u]:
            transpose[v].append(u)

    # Step 3: Second DFS in reverse order
    visited.clear()
    sccs = []

    def dfs2(node, component):
        visited.add(node)
        component.append(node)
        for neighbor in transpose[node]:
            if neighbor not in visited:
                dfs2(neighbor, component)

    for node in reversed(order):
        if node not in visited:
            component = []
            dfs2(node, component)
            sccs.append(component)

    return sccs
```

4. Articulation Points (Cut Vertices)

```
def find_articulation_points(graph):
    n = len(graph)
    visited = [False] * n
    disc = [0] * n  # Discovery time
    low = [0] * n   # Low value
    parent = [-1] * n
    ap = [False] * n  # Articulation points
    time = 0

    def dfs(u):
        nonlocal time
        children = 0
        visited[u] = True
        disc[u] = low[u] = time + 1
        time += 1
```

```
        for v in graph[u]:
            if not visited[v]:
                children += 1
                parent[v] = u
                dfs(v)

                # Update low value of u
                low[u] = min(low[u], low[v])

                # Check if u is articulation point
                if parent[u] == -1 and children > 1:  # Root with multiple
children
                    ap[u] = True
                if parent[u] != -1 and low[v] >= disc[u]:  # Non-root with
child that cannot reach ancestor
                    ap[u] = True

            elif v != parent[u]:  # Back edge
                low[u] = min(low[u], disc[v])

    for i in range(n):
        if not visited[i]:
            dfs(i)

    return [i for i in range(n) if ap[i]]
```

Mathematical Analysis

Time Complexity:

- **Adjacency List**: O(V + E) - each vertex and edge visited once
- **Adjacency Matrix**: O(V²) - must check all possible edges
- **Space Complexity**: O(V) for visited array and recursion stack

Properties:

- **Completeness**: Visits all reachable vertices from start node
- **Path Finding**: Can find path between two nodes
- **Depth-First Tree**: Forms a spanning tree of visited vertices
- **Edge Classification**:
  - **Tree Edge**: Edge in DFS tree
  - **Back Edge**: Edge to ancestor in DFS tree (indicates cycle)
  - **Forward Edge**: Edge to descendant not in DFS tree
  - **Cross Edge**: Edge between vertices in different subtrees

Correctness:

- **Invariant**: At any point, visited nodes form a connected component
- **Termination**: Algorithm terminates when all reachable nodes visited
- **Completeness**: If node v is reachable from u, DFS(u) will visit v

Applications in GATE Context

1. Graph Connectivity:

- **Connected Components**: Number of DFS calls needed to visit all nodes
- **Biconnected Components**: Using articulation points
- **Bridge Detection**: Edges whose removal increases components

2. Path Problems:

- **Existence of Path**: DFS can determine if path exists between nodes
- **All Paths**: Can enumerate all paths between nodes (with modifications)
- **Longest Path**: NP-hard in general graphs, but solvable in DAGs with DFS

3. Tree Problems:

- **Tree Traversal**: Preorder, inorder, postorder are DFS variants
- **Tree Diameter**: Two DFS passes can find diameter
- **Lowest Common Ancestor**: Can be found using DFS with binary lifting

4. Puzzle Solving:

- **Maze Solving**: DFS explores all paths systematically
- **Sudoku**: Backtracking with DFS
- **N-Queens**: DFS with pruning

Comparison with BFS:

Aspect	**DFS**	**BFS**
Data Structure	Stack	Queue
Space Complexity	O(V) worst case	O(V) worst case
Shortest Path	No (unweighted)	Yes (unweighted)
Memory Usage	Less for deep trees	Less for wide trees
Cycle Detection	Yes (directed/undirected)	Yes (undirected)
Topological Sort	Yes	No (needs modification)
Connected Components	Yes	Yes

GATE Tips:

- DFS uses recursion stack, can cause stack overflow for deep graphs
- Iterative DFS avoids recursion limits but needs explicit stack
- For cycle detection in directed graphs, need recursion stack tracking

- DFS tree edges reveal graph structure (back edges indicate cycles)
- Time complexity is always O(V + E) for adjacency list representation
- For disconnected graphs, need to call DFS for each unvisited node

1.9 Dijkstra's Algorithm (6)

Extended Comprehensive Coverage

Core Algorithm and Implementation

Algorithm:

```
import heapq

def dijkstra(graph, start):
    """
    graph: adjacency list {u: [(v, weight), ...]}
    start: source node
    Returns: (distances, predecessors)
    """
    # Initialize distances
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    predecessors = {node: None for node in graph}

    # Priority queue: (distance, node)
    pq = [(0, start)]
    visited = set()

    while pq:
        current_dist, current_node = heapq.heappop(pq)

        # Skip if already processed with better distance
        if current_node in visited:
            continue

        visited.add(current_node)

        # Process neighbors
        for neighbor, weight in graph[current_node]:
            if neighbor in visited:
                continue

            new_dist = current_dist + weight

            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
                predecessors[neighbor] = current_node
                heapq.heappush(pq, (new_dist, neighbor))
```

```
    return distances, predecessors
```

Variations and Optimizations

1. Dijkstra with Fibonacci Heap (Theoretical)

```
# Pseudocode - Fibonacci heap provides better amortized complexity
def dijkstra_fibonacci(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    fib_heap = FibonacciHeap()

    # Insert all nodes with initial distances
    nodes = {}
    for node in graph:
        nodes[node] = fib_heap.insert(node, distances[node])

    while not fib_heap.is_empty():
        current_node, current_dist = fib_heap.extract_min()

        for neighbor, weight in graph[current_node]:
            new_dist = current_dist + weight

            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
                fib_heap.decrease_key(nodes[neighbor], new_dist)

    return distances
```

2. Dijkstra for DAGs (Simplified)

```
def dijkstra_dag(graph, start):
    # For DAGs, can use topological sort for O(V+E) time
    topo_order = topological_sort(graph)
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    for node in topo_order:
        if distances[node] == float('inf'):
            continue

        for neighbor, weight in graph[node]:
            new_dist = distances[node] + weight
            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
```

```
    return distances
```

3. Bidirectional Dijkstra

```
def bidirectional_dijkstra(graph, start, target):
    # Run Dijkstra from start and target simultaneously
    forward_dist = {node: float('inf') for node in graph}
    backward_dist = {node: float('inf') for node in graph}
    forward_dist[start] = 0
    backward_dist[target] = 0

    forward_pq = [(0, start)]
    backward_pq = [(0, target)]
    forward_visited = set()
    backward_visited = set()

    best_path = float('inf')
    best_meeting_node = None

    while forward_pq and backward_pq:
        # Process forward search
        if forward_pq[0][0] <= backward_pq[0][0]:
            process_node(forward_pq, forward_dist, forward_visited, graph,
                         backward_dist, backward_visited, best_path,
best_meeting_node)
        else:
            process_node(backward_pq, backward_dist, backward_visited,
graph,
                         forward_dist, forward_visited, best_path,
best_meeting_node)

    return best_path if best_path != float('inf') else -1
```

Mathematical Analysis and Correctness

Time Complexity:

- **Binary Heap**: O((V + E) log V)
  - Each vertex inserted once: O(V log V)
  - Each edge may cause decrease-key: O(E log V)
- **Fibonacci Heap**: O(V log V + E) amortized
  - Insert and extract-min: O(log V) amortized
  - Decrease-key: O(1) amortized
- **Array Implementation**: O(V² + E)
  - Used when graph is dense (E ≈ V²)

Space Complexity: O(V + E)

- Distance array: O(V)
- Predecessor array: O(V)
- Priority queue: O(V)
- Graph representation: O(E)

Correctness Proof:

- **Lemma 1 (Greedy Choice)**: When vertex u is added to S (visited set), dist[u] = δ(s,u)
- **Lemma 2 (Optimal Substructure)**: Shortest path to u consists of shortest paths to intermediate vertices
- **Proof by Induction**:
  - **Base**: dist[s] = 0 = δ(s,s)
  - **Inductive Step**: Assume all vertices in S have correct distances. Let u be next vertex added. For any path from s to u through v ∉ S, dist[v] ≥ dist[u] (by heap property), so path through v cannot be shorter than dist[u]

Limitations:

- **Negative Weights**: Fails if any edge has negative weight
- **Negative Cycles**: Cannot detect negative cycles
- **All-Pairs**: Less efficient than Floyd-Warshall for dense graphs

Advanced Applications

1. A* Search Algorithm (Heuristic Extension)

```
def a_star(graph, start, goal, heuristic):
    """
    heuristic: function h(n) estimating cost from n to goal
    Must be admissible (never overestimates)
    """
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    f_scores = {node: float('inf') for node in graph}
    f_scores[start] = heuristic(start, goal)

    pq = [(f_scores[start], start)]
    predecessors = {node: None for node in graph}

    while pq:
        _, current = heapq.heappop(pq)

        if current == goal:
            return reconstruct_path(predecessors, start, goal)
```

```
        for neighbor, weight in graph[current]:
            tentative_g = distances[current] + weight

            if tentative_g < distances[neighbor]:
                predecessors[neighbor] = current
                distances[neighbor] = tentative_g
                f_scores[neighbor] = tentative_g + heuristic(neighbor, goal)

                if neighbor not in [n for _, n in pq]:
                    heapq.heappush(pq, (f_scores[neighbor], neighbor))

    return None
```

2. Dial's Algorithm (Integer Weights)

```
def dijkstra_dial(graph, start, max_weight):
    """
    For graphs with small integer weights (≤ C)
    Time complexity: O(V + E + VC)
    """
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Buckets for each possible distance value
    buckets = [[] for _ in range(max_weight * len(graph))]
    buckets[0].append(start)

    idx = 0  # Current bucket index
    visited = set()

    while idx < len(buckets):
        if not buckets[idx]:
            idx += 1
            continue

        node = buckets[idx].pop()
        if node in visited:
            continue

        visited.add(node)

        for neighbor, weight in graph[node]:
            new_dist = distances[node] + weight
            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
                buckets[new_dist].append(neighbor)

        if not buckets[idx]:
            idx += 1
```

```
    return distances
```

3. Contraction Hierarchies (Advanced Routing)

```
# Conceptual outline for modern routing algorithms
def preprocess_contraction_hierarchy(graph):
    """
    Preprocessing step for fast shortest path queries
    1. Order nodes by importance
    2. Contract nodes (remove and add shortcuts)
    3. Store hierarchy information
    """
    # This is a complex preprocessing step used in real navigation systems
    # Query time becomes O(log V) or better
    pass
```

GATE-Specific Insights

Common Question Types:

1. **Algorithm Execution**: Trace Dijkstra's algorithm on given graph
2. **Complexity Analysis**: Time/space complexity questions
3. **Data Structure Choice**: When to use which priority queue implementation
4. **Limitations**: Why Dijkstra fails with negative weights
5. **Comparisons**: Dijkstra vs Bellman-Ford vs Floyd-Warshall

Key Points to Remember:

- **Greedy Algorithm**: Always picks closest unvisited node
- **Non-negative Weights Only**: Critical constraint
- **Single Source**: Finds shortest paths from one source to all others
- **Priority Queue**: Essential for efficiency
- **Relaxation Step**: Key operation that updates distances

Comparison Table:
Algorithm	**Time Complexity**	**Negative Weights**	**Negative Cycles**	**Use Case**
Dijkstra	O((V+E)log V)	No	No	Non-negative weights
Bellman-Ford	O(VE)	Yes	Detects	General graphs, negative weights
Floyd-Warshall	O(V^3)	Yes	Detects	All-pairs shortest paths
SPFA	O(E) avg, O(VE) worst	Yes	Detects	Sparse graphs with negative weights

GATE Problem-Solving Strategy:

1. **Check Weights**: If any negative weights, Dijkstra cannot be used

2. **Identify Source**: Dijkstra is single-source algorithm
3. **Data Structure**: For dense graphs, array implementation may be better
4. **Trace Carefully**: When tracing, maintain priority queue state
5. **Termination**: Algorithm terminates when all reachable nodes processed

Sample GATE Questions:

1. What is time complexity of Dijkstra using binary heap?
   - Answer: O((V + E) log V)
2. Why does Dijkstra fail with negative weights?
   - Answer: Greedy choice property fails - a node may be processed before finding shorter path through negative edge
3. How many times is each edge relaxed in Dijkstra?
   - Answer: At most once (unlike Bellman-Ford which relaxes |V|-1 times)

1.10 Directed Graphs (Extended)

Core Concepts and Properties

Definition: A directed graph (digraph) G = (V, E) consists of:

- **Vertices (V)**: Set of nodes
- **Edges (E)**: Set of ordered pairs (u, v) where u, v ∈ V
- **Direction**: Edge (u, v) goes from u to v, not vice versa unless (v, u) exists

Key Properties:

- **Indegree**: Number of edges entering a vertex
- **Outdegree**: Number of edges leaving a vertex
- **Source**: Vertex with indegree 0
- **Sink**: Vertex with outdegree 0
- **Strongly Connected**: Path exists between every pair of vertices in both directions
- **Weakly Connected**: Connected when directions ignored

Important Algorithms for Directed Graphs

1. Topological Sorting

```
def topological_sort_kahn(graph):
    """
    Kahn's algorithm using BFS
    Returns topological order or empty list if cycle exists
    """
    # Calculate indegrees
    indegree = {node: 0 for node in graph}
    for u in graph:
```

```
        for v in graph[u]:
            indegree[v] = indegree.get(v, 0) + 1

    # Queue for nodes with indegree 0
    queue = deque([node for node in indegree if indegree[node] == 0])
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)

        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    # Check if all nodes processed (no cycle)
    if len(result) != len(graph):
        return []  # Cycle detected

    return result
```

2. Strongly Connected Components (Tarjan's Algorithm)

```
def tarjans_scc(graph):
    """
    Tarjan's algorithm using single DFS pass
    Returns list of SCCs
    """
    index_counter = [0]
    stack = []
    indices = {}
    lowlink = {}
    result = []

    def strongconnect(v):
        indices[v] = index_counter[0]
        lowlink[v] = index_counter[0]
        index_counter[0] += 1
        stack.append(v)

        for w in graph[v]:
            if w not in indices:
                strongconnect(w)
                lowlink[v] = min(lowlink[v], lowlink[w])
            elif w in stack:
                lowlink[v] = min(lowlink[v], indices[w])

        # If v is root node, pop stack to generate SCC
```

```
        if lowlink[v] == indices[v]:
            scc = []
            while True:
                w = stack.pop()
                scc.append(w)
                if w == v:
                    break
            result.append(scc)

    for v in graph:
        if v not in indices:
            strongconnect(v)

    return result
```

3. Transitive Closure (Floyd-Warshall)

```
def transitive_closure(graph):
    """
    Computes transitive closure using Floyd-Warshall
    Returns reachability matrix
    """
    n = len(graph)
    # Initialize reachability matrix
    reach = [[False] * n for _ in range(n)]

    # Set direct edges
    for i in range(n):
        reach[i][i] = True  # Self loops
        for j in graph[i]:
            reach[i][j] = True

    # Floyd-Warshall
    for k in range(n):
        for i in range(n):
            for j in range(n):
                reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])

    return reach
```

4. Shortest Paths in DAGs

```
def dag_shortest_paths(graph, start):
    """
    Single-source shortest paths in DAG
    Time complexity: O(V + E)
    """
    # Get topological order
```

```
    topo_order = topological_sort_kahn(graph)
    if not topo_order:
        raise ValueError("Graph has cycle, not a DAG")

    # Initialize distances
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Process nodes in topological order
    for node in topo_order:
        if distances[node] == float('inf'):
            continue

        for neighbor, weight in graph[node]:
            new_dist = distances[node] + weight
            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist

    return distances
```

Advanced Concepts and Applications

1. Dominators and Dominator Trees

```
def compute_dominators(graph, start):
    """
    Computes immediate dominators using Lengauer-Tarjan algorithm
    Dominator: Node d dominates node n if every path from start to n goes
through d
    """
    # This is a complex algorithm used in compiler optimization
    # Implementation omitted for brevity
    pass
```

2. Minimum Spanning Arborescence (Chu-Liu/Edmonds)

```
def min_spanning_arborescence(graph, root):
    """
    Finds minimum spanning tree directed away from root
    Time complexity: O(VE)
    """
    # Implementation involves cycle detection and contraction
    # Used in network design and parsing
    pass
```

3. PageRank Algorithm (Google)

```python
def pagerank(graph, damping=0.85, iterations=100):
    """
    Computes PageRank for directed graph
    graph: adjacency list where graph[u] = [v1, v2, ...] (outgoing links)
    """
    nodes = list(graph.keys())
    n = len(nodes)
    rank = {node: 1/n for node in nodes}

    # Precompute outdegrees
    outdegree = {node: len(graph[node]) for node in nodes}

    for _ in range(iterations):
        new_rank = {node: (1 - damping)/n for node in nodes}

        for node in nodes:
            if outdegree[node] == 0:  # Sink node
                continue

            contribution = rank[node] / outdegree[node]

            for neighbor in graph[node]:
                new_rank[neighbor] += damping * contribution

        rank = new_rank

    return rank
```

Mathematical Properties and Theorems

Euler's Theorem for Directed Graphs:

- A directed graph has an Euler circuit iff:
  - It is strongly connected
  - For every vertex: indegree = outdegree
- A directed graph has an Euler path iff:
  - At most one vertex has outdegree = indegree + 1 (start)
  - At most one vertex has indegree = outdegree + 1 (end)
  - All other vertices have equal indegree and outdegree
  - Graph is connected when directions ignored

Handshaking Lemma for Directed Graphs:

- Sum of all indegrees = Sum of all outdegrees = |E|

Dilworth's Theorem:

- In any finite partially ordered set, the size of the largest antichain equals the minimum number of chains needed to cover the set
- Application: Minimum path cover in DAGs

GATE-Specific Applications

1. Dependency Resolution:

- **Compiler Design**: Instruction scheduling, register allocation
- **Build Systems**: Make, Gradle dependency graphs
- **Database Transactions**: Serializability checking

2. Program Analysis:

- **Control Flow Graphs**: Basic blocks and edges represent program flow
- **Data Flow Analysis**: Reaching definitions, live variables
- **Call Graphs**: Function call relationships

3. Network Flow:

- **Flow Networks**: Source to sink flow with capacities
- **Bipartite Matching**: Can be reduced to max flow in directed graphs
- **Project Selection**: Choosing profitable projects with dependencies

4. Scheduling Problems:

- **Task Scheduling**: Precedence constraints form DAG
- **Course Prerequisites**: Topological sort for course planning
- **Manufacturing Processes**: Assembly line scheduling

GATE Problem-Solving Strategy:

1. **Identify Graph Type**: Determine if directed/undirected, cyclic/acyclic
2. **Check Properties**: Look for DAG properties, strong connectivity
3. **Algorithm Selection**:
   - **Topological Sort**: For DAGs with dependencies
   - **SCC Detection**: For strongly connected components
   - **Shortest Paths**: Dijkstra (non-negative) or Bellman-Ford (general)
   - **Transitive Closure**: For reachability queries
4. **Complexity Analysis**: Consider graph density (sparse vs dense)
5. **Edge Cases**: Handle disconnected graphs, self-loops, multiple edges

Common GATE Questions:

1. **Topological Sort**: Given DAG, find valid topological order
2. **Cycle Detection**: Determine if directed graph has cycle

3. **SCC Count**: Find number of strongly connected components
4. **Shortest Path**: Compute shortest path in DAG or general directed graph
5. **Transitive Closure**: Determine if path exists between nodes

Key Formulas:

- **Maximum Edges in DAG**: n(n-1)/2 (complete DAG)
- **Minimum Edges for Strong Connectivity**: n (cycle)
- **Topological Sort Existence**: iff graph is acyclic
- **SCC Property**: Each SCC forms a maximal strongly connected subgraph

1.11 Double Hashing (1)

Key Concepts: $h(k,i) = h_1(k) + icdoth_2(k)bmodm.$

1.12 Dynamic Programming (9)

Key Concepts: Dynamic Programming (DP) is a powerful algorithmic paradigm for solving optimization problems by breaking them into overlapping subproblems. It trades space for time by storing solutions to avoid recomputation.

Fundamental Principles:

1. Optimal Substructure:

- Optimal solution to problem contains optimal solutions to subproblems
- Can construct optimal solution from optimal solutions of subproblems
- **Example**: Shortest path - if P is shortest path from A to C via B, then subpaths A→B and B→C are also shortest

2. Overlapping Subproblems:

- Same subproblems are solved multiple times in naive recursive approach
- DP stores solutions to avoid recomputation
- **Example**: Fibonacci - F(n) needs F(n-1) and F(n-2), F(n-1) also needs F(n-2)

3. Memoization vs Tabulation:

- **Memoization (Top-down)**: Recursive approach with caching
- **Tabulation (Bottom-up)**: Iterative approach filling table

When to Use DP:

1. Problem has optimal substructure
2. Problem has overlapping subproblems
3. Problem asks for optimization (min/max/count)
4. Brute force has exponential time complexity

DP vs Divide & Conquer:

- **D&C**: Subproblems are independent (merge sort, quick sort)
- **DP**: Subproblems overlap and share solutions

Classic DP Problems:

1. Fibonacci Sequence:

Naive Recursion: $T(n) = T(n-1) + T(n-2) + O(1) = O(2^n)$

```
int fibRecursive(int n) {
    if (n <= 1) return n;
    return fibRecursive(n-1) + fibRecursive(n-2);
}
```

Memoization: $O(n)$ time, $O(n)$ space

```
int fibMemo(int n, vector<int>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];

    memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);
    return memo[n];
}
```

Tabulation: $O(n)$ time, $O(n)$ space

```
int fibDP(int n) {
    if (n <= 1) return n;

    vector<int> dp(n+1);
    dp[0] = 0; dp[1] = 1;

    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];

    return dp[n];
}
```

Space Optimized: $O(n)$ time, $O(1)$ space

```
int fibOptimized(int n) {
    if (n <= 1) return n;

    int prev2 = 0, prev1 = 1, curr;
    for (int i = 2; i <= n; i++) {
```

```
        curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return curr;
}
```

2. Longest Common Subsequence (LCS):

Problem: Find length of longest subsequence common to two strings

State Definition: `dp[i][j]` = LCS length of `X[0...i-1]` and `Y[0...j-1]`

Recurrence:

```
if X[i-1] == Y[j-1]:
    dp[i][j] = dp[i-1][j-1] + 1
else:
    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

Implementation: $O(mn)$ time, $O(mn)$ space

```
int LCS(string X, string Y) {
    int m = X.length(), n = Y.length();
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i-1] == Y[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }

    return dp[m][n];
}
```

Space Optimized: $O(mn)$ time, $O(\min(m, n))$ space

```
int LCSOptimized(string X, string Y) {
    int m = X.length(), n = Y.length();
    if (m < n) { swap(X, Y); swap(m, n); }  // Ensure X is longer

    vector<int> prev(n+1, 0), curr(n+1, 0);

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
```

```
            if (X[i-1] == Y[j-1])
                curr[j] = prev[j-1] + 1;
            else
                curr[j] = max(prev[j], curr[j-1]);
        }
        prev = curr;
    }

    return curr[n];
}
```

3. 0/1 Knapsack Problem:

Problem: Given items with weights and values, maximize value within weight capacity

State: `dp[i][w]` = maximum value using first `i` items with capacity `w`

Recurrence:

```
if weight[i-1] <= w:
    dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i-1]] + value[i-1])
else:
    dp[i][w] = dp[i-1][w]
```

Implementation: $O(nW)$ time, $O(nW)$ space

```
int knapsack(vector<int>& weights, vector<int>& values, int W) {
    int n = weights.size();
    vector<vector<int>> dp(n+1, vector<int>(W+1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= W; w++) {
            if (weights[i-1] <= w)
                dp[i][w] = max(dp[i-1][w],
                               dp[i-1][w-weights[i-1]] + values[i-1]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }

    return dp[n][W];
}
```

Space Optimized: $O(nW)$ time, $O(W)$ space

```
int knapsackOptimized(vector<int>& weights, vector<int>& values, int W) {
    vector<int> dp(W+1, 0);
```

```
    for (int i = 0; i < weights.size(); i++) {
        for (int w = W; w >= weights[i]; w--) {  // Reverse order!
            dp[w] = max(dp[w], dp[w-weights[i]] + values[i]);
        }
    }

    return dp[W];
}
```

4. Matrix Chain Multiplication:

Problem: Find optimal parenthesization to minimize scalar multiplications

State: `dp[i][j]` = minimum multiplications for matrices A_i to A_j

Recurrence: $dp[i][j] = \min_{i \leq k < j}(dp[i][k] + dp[k+1][j] + p_i \times p_{k+1} \times p_{j+1})$

Implementation: $O(n^3)$ time, $O(n^2)$ space

```
int matrixChainMultiplication(vector<int>& p) {
    int n = p.size() - 1;  // Number of matrices
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // l is chain length
    for (int l = 2; l <= n; l++) {
        for (int i = 0; i < n - l + 1; i++) {
            int j = i + l - 1;
            dp[i][j] = INT_MAX;

            for (int k = i; k < j; k++) {
                int cost = dp[i][k] + dp[k+1][j] + p[i] * p[k+1] * p[j+1];
                dp[i][j] = min(dp[i][j], cost);
            }
        }
    }

    return dp[0][n-1];
}
```

5. Edit Distance (Levenshtein Distance):

Problem: Minimum operations (insert, delete, substitute) to transform one string to another

State: `dp[i][j]` = edit distance between first `i` characters of string1 and first `j` characters of string2

Implementation: $O(mn)$ time, $O(mn)$ space

```cpp
int editDistance(string str1, string str2) {
    int m = str1.length(), n = str2.length();
    vector<vector<int>> dp(m+1, vector<int>(n+1));

    // Base cases
    for (int i = 0; i <= m; i++) dp[i][0] = i;  // Delete all
    for (int j = 0; j <= n; j++) dp[0][j] = j;  // Insert all

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];  // No operation needed
            else
                dp[i][j] = 1 + min({dp[i-1][j],    // Delete
                                    dp[i][j-1],    // Insert
                                    dp[i-1][j-1]});// Substitute
        }
    }

    return dp[m][n];
}
```

6. Longest Increasing Subsequence (LIS):

Naive DP: $O(n^2)$ time, $O(n)$ space

```cpp
int LIS_DP(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n, 1);  // dp[i] = LIS ending at index i

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[j] < arr[i])
                dp[i] = max(dp[i], dp[j] + 1);
        }
    }

    return *max_element(dp.begin(), dp.end());
}
```

Optimized with Binary Search: $O(n \log n)$ time, $O(n)$ space

```cpp
int LIS_Optimized(vector<int>& arr) {
    vector<int> tails;  // tails[i] = smallest tail of LIS of length i+1

    for (int num : arr) {
        auto it = lower_bound(tails.begin(), tails.end(), num);
```

```
            if (it == tails.end())
                tails.push_back(num);
            else
                *it = num;
        }

        return tails.size();
    }
```

Advanced DP Patterns:

1. Interval DP: Problems on ranges/intervals

- **Template**: `dp[i][j]` represents optimal solution for range `[i, j]`
- **Examples**: Matrix chain multiplication, palindrome partitioning

2. Tree DP: Problems on trees

- **Template**: `dp[node][state]` represents solution for subtree rooted at node
- **Examples**: Tree diameter, maximum independent set

3. Digit DP: Count numbers with certain digit properties

- **State**: Current position, tight constraint, previous digits
- **Examples**: Count numbers with sum of digits = K

4. Bitmask DP: State includes subset information

- **Template**: `dp[mask]` where mask represents subset
- **Examples**: Traveling Salesman Problem, assignment problems

5. DP on Grids: Path problems on 2D grids

- **Template**: `dp[i][j]` represents solution reaching cell (i,j)
- **Examples**: Unique paths, minimum path sum

Problem-Solving Strategy:

Step 1: Identify DP Structure

- Does problem have optimal substructure?
- Are there overlapping subproblems?
- Is it asking for optimization (min/max/count)?

Step 2: Define State

- What parameters uniquely identify a subproblem?
- What's the minimum information needed?

- How many dimensions needed?

Step 3: Write Recurrence

- How does current state relate to previous states?
- What are the transitions?
- What are the base cases?

Step 4: Determine Order

- For tabulation: What order to fill the table?
- Ensure dependencies are computed first

Step 5: Optimize

- Can space complexity be reduced?
- Can time complexity be improved?
- Are there redundant computations?

Space Optimization Techniques:

1. Rolling Array: When current state depends only on previous row/column

```
// Instead of dp[i][j], use dp[j] and prev[j]
vector<int> prev(n), curr(n);
```

2. In-place Updates: When safe to overwrite previous values

```
// Process in reverse order to avoid overwriting needed values
for (int i = n-1; i >= 0; i--)
```

3. State Compression: Reduce number of states

```
// Use mathematical properties to reduce dimensions
```

Common Mistakes and GATE Traps:

1. **Incorrect State Definition**: Not capturing all necessary information
2. **Wrong Base Cases**: Forgetting edge cases or boundary conditions
3. **Order of Computation**: Computing states before their dependencies
4. **Space Optimization Errors**: Overwriting values still needed
5. **Integer Overflow**: Not handling large intermediate values
6. **Time Complexity Analysis**: Miscounting states or transitions

GATE Tips:

- Time complexity = (Number of states) × (Time per transition)
- Space complexity often reducible by one dimension
- Memoization has recursion overhead, tabulation doesn't
- Always check base cases and boundary conditions
- For optimization problems, think about what you're optimizing
- Common patterns: Linear DP, Interval DP, Tree DP, Bitmask DP
- Practice identifying when DP is applicable vs when it's not

Comparison with Other Techniques:

Technique	Time	Space	When to Use
Recursion	Exponential	O(depth)	Small inputs, clear structure
Memoization	Polynomial	O(states)	Natural recursive structure
Tabulation	Polynomial	O(states)	Iterative approach preferred
Greedy	O(n log n)	O(1)	Greedy choice property holds

Practice Problems by Category:

Beginner: Fibonacci, Climbing Stairs, House Robber
Intermediate: LCS, Edit Distance, Coin Change, Knapsack
Advanced: Matrix Chain, Palindrome Partitioning, Burst Balloons
Expert: Digit DP, Tree DP, Bitmask DP problems

1.13 Graph Algorithms (11)

(See general graph).

1.14 Graph Search (22)

(See DFS, BFS: $O(V + E)$).

1.15 Greedy Algorithms (5)

Key Concepts: Greedy algorithms make locally optimal choices at each step, hoping to find a global optimum. They work when the greedy choice property and optimal substructure hold.

Fundamental Principles:

1. Greedy Choice Property:

- Locally optimal choice leads to globally optimal solution
- Can make choice without considering future consequences
- Once made, choice is never reconsidered

2. Optimal Substructure:

- Optimal solution contains optimal solutions to subproblems
- After making greedy choice, remaining problem has same property

3. No Backtracking:

- Decisions are final and irreversible
- Contrast with dynamic programming (considers all possibilities)
- Contrast with backtracking (explores and backtracks)

When Greedy Works:

- Problem exhibits greedy choice property
- Problem has optimal substructure
- Local optimum leads to global optimum
- **Matroid structure** (advanced): Mathematical framework guaranteeing greedy optimality

Classic Greedy Algorithms:

1. Activity Selection Problem:

Problem: Select maximum number of non-overlapping activities

Greedy Strategy: Always pick activity that finishes earliest

Algorithm: $O(n \log n)$ time

```
int activitySelection(vector<pair<int,int>>& activities) {
    // Sort by finish time
    sort(activities.begin(), activities.end(),
         [](const pair<int,int>& a, const pair<int,int>& b) {
             return a.second < b.second;
         });

    int count = 1;  // First activity always selected
    int lastFinish = activities[0].second;

    for (int i = 1; i < activities.size(); i++) {
        if (activities[i].first >= lastFinish) {
            count++;
            lastFinish = activities[i].second;
        }
    }

    return count;
}
```

Proof of Correctness:

- Let A = greedy solution, O = optimal solution
- If A ≠ O, let first difference be at position i
- Replace O's choice with A's choice (earlier finish time)
- This doesn't reduce optimality, contradiction

2. Fractional Knapsack:

Problem: Maximize value with weight constraint, items can be fractioned

Greedy Strategy: Sort by value-to-weight ratio, take highest ratios first

Algorithm: $O(n \log n)$ time

```
double fractionalKnapsack(vector<pair<int,int>>& items, int W) {
    // Create value-to-weight ratio pairs
    vector<pair<double, int>> ratios;
    for (int i = 0; i < items.size(); i++) {
        ratios.push_back({(double)items[i].first / items[i].second, i});
    }

    // Sort by ratio in descending order
    sort(ratios.rbegin(), ratios.rend());

    double totalValue = 0;
    int remainingWeight = W;

    for (auto& ratio : ratios) {
        int idx = ratio.second;
        int weight = items[idx].second;
        int value = items[idx].first;

        if (weight <= remainingWeight) {
            // Take entire item
            totalValue += value;
            remainingWeight -= weight;
        } else {
            // Take fraction of item
            totalValue += value * ((double)remainingWeight / weight);
            break;
        }
    }

    return totalValue;
}
```

Note: 0/1 Knapsack cannot be solved greedily (requires DP)

3. Huffman Coding:

Problem: Optimal prefix-free binary codes for data compression

Greedy Strategy: Always merge two nodes with smallest frequencies

Algorithm: $O(n \log n)$ time

```
struct Node {
    char ch;
    int freq;
    Node* left;
    Node* right;

    Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq;  // Min heap
    }
};

Node* huffmanCoding(vector<pair<char, int>>& frequencies) {
    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Create leaf nodes
    for (auto& p : frequencies) {
        pq.push(new Node(p.first, p.second));
    }

    // Build Huffman tree
    while (pq.size() > 1) {
        Node* right = pq.top(); pq.pop();
        Node* left = pq.top(); pq.pop();

        Node* merged = new Node('\0', left->freq + right->freq);
        merged->left = left;
        merged->right = right;

        pq.push(merged);
    }

    return pq.top();  // Root of Huffman tree
}
```

Properties:

- Optimal prefix-free code

- More frequent characters get shorter codes
- Average code length is minimized

4. Minimum Spanning Tree (MST):

Kruskal's Algorithm: $O(E \log E)$ time

```
struct Edge {
    int u, v, weight;
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

class UnionFind {
    vector<int> parent, rank;
public:
    UnionFind(int n) : parent(n), rank(n, 0) {
        iota(parent.begin(), parent.end(), 0);
    }

    int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);  // Path compression
        return parent[x];
    }

    bool unite(int x, int y) {
        int px = find(x), py = find(y);
        if (px == py) return false;

        if (rank[px] < rank[py]) swap(px, py);
        parent[py] = px;
        if (rank[px] == rank[py]) rank[px]++;
        return true;
    }
};

vector<Edge> kruskalMST(vector<Edge>& edges, int V) {
    sort(edges.begin(), edges.end());

    UnionFind uf(V);
    vector<Edge> mst;

    for (Edge& e : edges) {
        if (uf.unite(e.u, e.v)) {
            mst.push_back(e);
            if (mst.size() == V - 1) break;
        }
```

```
    }

    return mst;
}
```

Prim's Algorithm: $O(E \log V)$ time with binary heap

```
vector<pair<int,int>> primMST(vector<vector<pair<int,int>>>& graph) {
    int V = graph.size();
    vector<bool> inMST(V, false);
    vector<int> key(V, INT_MAX);
    vector<int> parent(V, -1);

    priority_queue<pair<int,int>, vector<pair<int,int>>,
greater<pair<int,int>>> pq;

    key[0] = 0;
    pq.push({0, 0});  // {weight, vertex}

    vector<pair<int,int>> mst;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        if (inMST[u]) continue;
        inMST[u] = true;

        if (parent[u] != -1)
            mst.push_back({parent[u], u});

        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;

            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
                pq.push({weight, v});
            }
        }
    }

    return mst;
}
```

5. Dijkstra's Shortest Path:

Problem: Single-source shortest paths with non-negative weights

Greedy Strategy: Always process vertex with minimum distance

Algorithm: $O((V + E) \log V)$ time with binary heap

```
vector<int> dijkstra(vector<vector<pair<int,int>>>& graph, int src) {
    int V = graph.size();
    vector<int> dist(V, INT_MAX);
    priority_queue<pair<int,int>, vector<pair<int,int>>,
greater<pair<int,int>>> pq;

    dist[src] = 0;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        int d = pq.top().first;
        pq.pop();

        if (d > dist[u]) continue;  // Already processed

        for (auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

6. Job Scheduling Problems:

Shortest Job First (SJF):

```
double shortestJobFirst(vector<int>& jobs) {
    sort(jobs.begin(), jobs.end());

    int totalTime = 0, waitTime = 0;
    for (int job : jobs) {
        waitTime += totalTime;
        totalTime += job;
    }
```

```
    return (double)waitTime / jobs.size();
}
```

Job Scheduling with Deadlines:

```
int jobScheduling(vector<tuple<int,int,int>>& jobs) {  // {profit, deadline, id}
    sort(jobs.rbegin(), jobs.rend());  // Sort by profit descending

    int maxDeadline = 0;
    for (auto& job : jobs)
        maxDeadline = max(maxDeadline, get<1>(job));

    vector<bool> slot(maxDeadline + 1, false);
    int totalProfit = 0;

    for (auto& job : jobs) {
        int profit = get<0>(job);
        int deadline = get<1>(job);

        // Find latest available slot
        for (int i = deadline; i >= 1; i--) {
            if (!slot[i]) {
                slot[i] = true;
                totalProfit += profit;
                break;
            }
        }
    }

    return totalProfit;
}
```

When Greedy Fails:

1. 0/1 Knapsack: Greedy by value/weight ratio doesn't work

- **Counterexample**: Items {(60,10), (100,20), (120,30)}, capacity 50
- Greedy: Take (60,10) and (100,20), value = 160
- Optimal: Take (100,20) and (120,30), value = 220

2. Longest Path: Greedy doesn't work for longest path

- Always choose locally longest edge
- May lead to dead end, missing globally optimal path

3. Graph Coloring: Greedy coloring not always optimal

- May use more colors than necessary
- Optimal coloring is NP-hard

Matroid Theory (Advanced):

Definition: Mathematical structure (S, I) where:

- S is finite set (ground set)
- I is collection of subsets (independent sets) satisfying:
  1. $\emptyset \in I$
  2. If $A \in I$ and $B \subseteq A$, then $B \in I$
  3. If $A, B \in I$ and $|A| < |B|$, then $\exists x \in B \setminus A$ such that $A \cup \{x\} \in I$

Greedy Algorithm on Matroids:

- Sort elements by weight
- Greedily add elements that maintain independence
- **Theorem**: This gives optimal solution

Examples of Matroids:

- **Graphic Matroid**: Independent sets are forests
- **Uniform Matroid**: Independent sets have size ≤ k
- **Partition Matroid**: Elements partitioned, limited per partition

Problem-Solving Strategy:

Step 1: Identify Greedy Choice

- What local decision should we make?
- What criterion determines the "best" choice?
- Is this choice safe (won't prevent optimal solution)?

Step 2: Prove Greedy Choice Property

- Show that local optimum leads to global optimum
- Often by exchange argument or contradiction

Step 3: Prove Optimal Substructure

- After greedy choice, remaining problem has same structure
- Optimal solution to original = greedy choice + optimal solution to subproblem

Step 4: Implement and Analyze

- Usually involves sorting: $O(n \log n)$
- Main loop: $O(n)$ to $O(n^2)$ depending on operations

Comparison with Other Paradigms:

Paradigm	Time	Space	Optimality	When to Use
Greedy	O(n log n)	O(1)	Sometimes	Greedy choice property
DP	O(n²) - O(n³)	O(n) - O(n²)	Always	Overlapping subproblems
Backtracking	Exponential	O(depth)	Always	Small search space
Divide & Conquer	O(n log n)	O(log n)	Always	Independent subproblems

GATE Tips:

- Greedy works when local optimum = global optimum
- Common greedy strategies: earliest deadline, highest ratio, shortest/longest first
- MST algorithms (Kruskal, Prim) are classic greedy examples
- Dijkstra works only with non-negative weights
- Activity selection: sort by finish time, not duration
- Fractional knapsack: greedy works, 0/1 knapsack: DP needed
- Huffman coding: merge smallest frequencies first
- Job scheduling: sort by profit/deadline ratio
- Always prove correctness: greedy choice + optimal substructure

Common Mistakes:

1. **Assuming greedy works**: Not all optimization problems have greedy solutions
2. **Wrong greedy choice**: Choosing wrong local criterion
3. **Not proving correctness**: Greedy intuition can be misleading
4. **Ignoring constraints**: Greedy choice must respect problem constraints
5. **Incorrect sorting**: Wrong sorting criterion leads to wrong solution

Practice Problems:

- **Easy**: Activity selection, fractional knapsack, coin change (specific denominations)
- **Medium**: Job scheduling, Huffman coding, gas station problem
- **Hard**: Minimum number of platforms, candy distribution, jump game

1.16 Hashing (7)

Key Concepts: Hashing provides average-case O(1) access time by mapping keys to array indices using hash functions. Critical for implementing dictionaries, sets, and caches efficiently.

Hash Table Components:

1. Hash Function: $h : U \to \{0, 1, 2, \ldots, m-1\}$

- Maps keys from universe U to table indices
- Should distribute keys uniformly across table
- Should be fast to compute

2. Hash Table: Array of size m (table size)

- Each slot can store key-value pairs
- m should be chosen carefully (often prime)

3. Load Factor: $\alpha = \frac{n}{m}$

- n = number of elements stored
- m = table size
- Critical parameter affecting performance
- Keep $\alpha < 1$ for open addressing, $\alpha < 0.75$ typically

Hash Function Design:

Properties of Good Hash Functions:

1. **Uniform Distribution**: Each slot equally likely
2. **Fast Computation**: O(1) time to compute
3. **Deterministic**: Same key always maps to same slot
4. **Avalanche Effect**: Small key changes cause large hash changes

Common Hash Functions:

1. Division Method: $h(k) = k \bmod m$

- Simple and fast
- m should be prime, not close to power of 2
- Avoid $m = 2^p$ (only uses low-order bits)

2. Multiplication Method: $h(k) = \lfloor m(kA \bmod 1) \rfloor$

- A is constant, $0 < A < 1$
- Good choice: $A = \frac{\sqrt{5}-1}{2} \approx 0.618$ (golden ratio)
- m can be power of 2

3. Universal Hashing: $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$

- p is prime larger than universe size
- a, b chosen randomly from $\{1, 2, \ldots, p-1\}$ and $\{0, 1, \ldots, p-1\}$
- Guarantees good average-case performance

4. String Hashing:

```
int hashString(string s, int m) {
    int hash = 0;
    int p = 31;  // Prime number
    int p_pow = 1;

    for (char c : s) {
        hash = (hash + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }

    return hash;
}
```

Collision Resolution:

Collision: When $h(k_1) = h(k_2)$ for $k_1 \neq k_2$

1. Chaining (Separate Chaining):

Method: Each table slot contains linked list of colliding elements

Implementation:

```
class HashTableChaining {
    vector<list<pair<int, int>>> table;
    int size;

public:
    HashTableChaining(int m) : size(m) {
        table.resize(m);
    }

    int hash(int key) {
        return key % size;
    }

    void insert(int key, int value) {
        int index = hash(key);

        // Check if key already exists
        for (auto& pair : table[index]) {
            if (pair.first == key) {
                pair.second = value;
                return;
            }
        }

        table[index].push_back({key, value});
    }
```

```
    bool search(int key, int& value) {
        int index = hash(key);

        for (auto& pair : table[index]) {
            if (pair.first == key) {
                value = pair.second;
                return true;
            }
        }

        return false;
    }

    void remove(int key) {
        int index = hash(key);
        table[index].remove_if([key](const pair<int,int>& p) {
            return p.first == key;
        });
    }
};
```

Analysis:

- **Average case**: O(1 + α) for search/insert/delete
- **Worst case**: O(n) if all keys hash to same slot
- **Space**: O(m + n)
- **Load factor**: Can exceed 1

2. Open Addressing:

Method: All elements stored in table itself, use probing to resolve collisions

Probing Sequences:

Linear Probing: $h(k, i) = (h'(k) + i) \bmod m$

- Check slots $h'(k), h'(k) + 1, h'(k) + 2, \ldots$
- **Primary clustering**: Consecutive occupied slots

```
class HashTableLinearProbing {
    vector<pair<int, int>> table;
    vector<bool> occupied;
    int size, count;

public:
    HashTableLinearProbing(int m) : size(m), count(0) {
        table.resize(m);
```

```
        occupied.resize(m, false);
    }

    int hash(int key) {
        return key % size;
    }

    void insert(int key, int value) {
        if (count >= size * 0.75) resize();  // Maintain load factor

        int index = hash(key);

        while (occupied[index]) {
            if (table[index].first == key) {
                table[index].second = value;
                return;
            }
            index = (index + 1) % size;
        }

        table[index] = {key, value};
        occupied[index] = true;
        count++;
    }

    bool search(int key, int& value) {
        int index = hash(key);

        while (occupied[index]) {
            if (table[index].first == key) {
                value = table[index].second;
                return true;
            }
            index = (index + 1) % size;
        }

        return false;
    }
};
```

Quadratic Probing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

- Common: $c_1 = c_2 = 1/2$
- Reduces primary clustering
- **Secondary clustering**: Keys with same initial hash follow same probe sequence

Double Hashing: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

- $h_2(k)$ must be relatively prime to m

- Good choice: $h_2(k) = 7 - (k \bmod 7)$
- Eliminates clustering

Analysis of Open Addressing:

Assumptions: Uniform hashing, load factor $\alpha = n/m < 1$

Expected probe count:

- **Successful search**: $\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$
- **Unsuccessful search**: $\frac{1}{1-\alpha}$

For α = 0.5: ~1.4 probes (successful), ~2 probes (unsuccessful)
For α = 0.75: ~1.8 probes (successful), ~4 probes (unsuccessful)
For α = 0.9: ~2.6 probes (successful), ~10 probes (unsuccessful)

Dynamic Resizing:

When to Resize:

- **Chaining**: When α > threshold (e.g., 1.0)
- **Open addressing**: When α > threshold (e.g., 0.75)

Resizing Process:

1. Create new table with size ~2m (next prime)
2. Rehash all existing elements
3. Update table size

Amortized Analysis: O(1) average time per operation

Perfect Hashing:

Goal: Guarantee O(1) worst-case lookup time

Two-Level Scheme:

1. **First level**: Hash n keys into m = n slots
2. **Second level**: For each slot with collisions, use perfect hash table

Properties:

- **Space**: O(n)
- **Lookup time**: O(1) worst-case
- **Construction time**: O(n) expected
- **Static**: No insertions/deletions after construction

Cuckoo Hashing:

Method: Use two hash functions and two tables

- Each key stored in exactly one location
- If collision, "kick out" existing key to its alternate location

Properties:

- **Lookup**: O(1) worst-case (check at most 2 locations)
- **Insert**: O(1) expected, may require rehashing
- **Space**: ~50% utilization

Bloom Filters:

Purpose: Space-efficient probabilistic data structure for set membership

Properties:

- **False positives**: Possible
- **False negatives**: Never
- **Space**: Much smaller than hash table

Implementation:

```
class BloomFilter {
    vector<bool> bits;
    int m, k;  // m = bit array size, k = number of hash functions

public:
    BloomFilter(int size, int numHashes) : m(size), k(numHashes) {
        bits.resize(m, false);
    }

    void insert(int key) {
        for (int i = 0; i < k; i++) {
            int hash = (key + i * 7) % m;  // Simple hash family
            bits[hash] = true;
        }
    }

    bool mightContain(int key) {
        for (int i = 0; i < k; i++) {
            int hash = (key + i * 7) % m;
            if (!bits[hash]) return false;
        }
        return true;  // Might be false positive
    }
};
```

Applications:

- Web crawlers (avoid revisiting URLs)
- Database query optimization
- Network routers (packet filtering)
- Distributed systems (reduce network calls)

Hash Table Applications:

1. Symbol Tables: Compilers, interpreters
2. Caches: CPU caches, web caches, memoization
3. Databases: Hash indexes, hash joins
4. Cryptography: Hash functions, digital signatures
5. Data Deduplication: Identify duplicate files/data
6. Load Balancing: Consistent hashing
7. Distributed Systems: DHTs (Distributed Hash Tables)

Consistent Hashing:

Problem: Distribute keys across changing set of servers

Solution: Hash both keys and servers onto circle

- Key assigned to first server clockwise
- Adding/removing server affects only nearby keys

Properties:

- **Minimal disruption**: O(K/N) keys move when server added/removed
- **Load balancing**: Virtual nodes improve distribution

Cryptographic Hash Functions:

Properties:

1. **Deterministic**: Same input → same output
2. **Fast computation**: Efficient to compute
3. **Avalanche effect**: Small input change → large output change
4. **Pre-image resistance**: Hard to find input for given output
5. **Collision resistance**: Hard to find two inputs with same output

Examples: MD5 (broken), SHA-1 (deprecated), SHA-256, SHA-3

Performance Comparison:

Method	Average Search	Worst Search	Space	Notes
Chaining	O(1 + α)	O(n)	O(m + n)	Simple, handles high load
Linear Probing	O(1)	O(n)	O(m)	Cache-friendly, clustering
Quadratic Probing	O(1)	O(n)	O(m)	Less clustering
Double Hashing	O(1)	O(n)	O(m)	Best open addressing
Cuckoo	O(1)	O(1)	O(m)	Worst-case guarantee

Problem-Solving Strategy:

1. Choose Hash Function:

- Division method for general keys
- Multiplication method for unknown key distribution
- Universal hashing for adversarial inputs
- Cryptographic hash for security

2. Choose Collision Resolution:

- **Chaining**: Simple, handles high load factors
- **Open addressing**: Better cache performance, lower memory
- **Cuckoo**: Worst-case guarantees needed

3. Set Table Size:

- Prime number for division method
- Power of 2 for multiplication method
- Consider load factor requirements

4. Handle Dynamic Resizing:

- Monitor load factor
- Resize when threshold exceeded
- Rehash all elements

GATE Tips:

- Load factor $\alpha = n/m$ affects performance significantly
- Chaining allows $\alpha > 1$, open addressing requires $\alpha < 1$
- Universal hashing provides theoretical guarantees

- Perfect hashing gives O(1) worst-case but is static
- Bloom filters: false positives possible, false negatives never
- Consistent hashing minimizes disruption in distributed systems
- Cryptographic hashes provide security properties
- Linear probing suffers from primary clustering
- Double hashing eliminates clustering in open addressing
- Expected probe count in open addressing: 1/(1-α) for unsuccessful search

Common Mistakes:

1. **Poor hash function**: Using non-prime modulus, ignoring distribution
2. **High load factor**: Performance degrades significantly
3. **No resizing**: Table becomes inefficient as it fills
4. **Incorrect probing**: Off-by-one errors in probe sequences
5. **Deletion in open addressing**: Need tombstones or special handling
6. **Security issues**: Using non-cryptographic hash for security applications

1.17 Huffman Code (6)

Key Concepts: Prefix codes, greedy tree. **Algorithm**: Min-heap merge; avg length $sump_il_i$.

1.18 Identify Function (38)

Key Concepts: Identity in algos, e.g., no-op.

1.19 Insertion Sort (3)

Key Concepts: Build sorted prefix; $O(n^2)$.

1.20 Linear Probing (1)

Key Concepts: $h(k,i) = (h(k) + i)bmodm$; clustering issue.

1.21 Matrix Chain Ordering (3)

Key Concepts: DP for parenthesization; cost $minsump_ip_{i+1}cdots$. **Tips**: Table $m[i,j] = min_k(m[i,k] + m[k+1,j] + p_ip_{k+1}p_{j+1})$.

1.22 Merge Sort (3)

Key Concepts: Divide, sort, merge; $O(nlogn)$.

1.23 Merging (2)

Key Concepts: Two pointers for sorted lists.

1.24 Minimum Spanning Tree (34)

Key Concepts: Tree connecting all vertices with minimum total edge weight. Greedy algorithms produce optimal MST.

MST Properties:

- Exactly $|V| - 1$ edges
- No cycles (it's a tree)
- Connects all vertices (spanning)
- May not be unique if edge weights are equal
- Removing any edge disconnects the tree
- Adding any edge creates exactly one cycle

Cut Property: For any cut (S, V-S), the minimum weight edge crossing the cut is in some MST.

Cycle Property: For any cycle, the maximum weight edge is not in any MST.

Kruskal's Algorithm:

```
Kruskal(G):
    Sort all edges by weight (non-decreasing)
    MST = empty set
    For each edge (u,v) in sorted order:
        If u and v in different components:  // Use Union-Find
            Add (u,v) to MST
            Union(u, v)
    Return MST
```

- **Time Complexity**: $O(E \log E)$ or $O(E \log V)$ (sorting dominates)
- **Space**: $O(V)$ for Union-Find
- **Strategy**: Edge-centric, builds forest that merges into tree
- **Best for**: Sparse graphs (few edges)

Union-Find (Disjoint Set Union):

- **Find(x)**: Find root/representative of x's set
- **Union(x,y)**: Merge sets containing x and y
- **Optimizations**:
  - Path compression: Make nodes point directly to root
  - Union by rank/size: Attach smaller tree to larger
- **Complexity**: $O(\alpha(n))$ per operation (nearly constant, inverse Ackermann)

Prim's Algorithm:

```
Prim(G, start):
    MST = {start}
    Priority queue Q with edges from start
    While Q not empty and |MST| < V:
        (u,v) = Q.extractMin()  // Minimum weight edge
        If v not in MST:
            Add (u,v) to MST
            Add v to MST
            Add all edges from v to Q
    Return MST
```

- **Time Complexity**:
  - Binary heap: $O((V + E) \log V)$ = $O(E \log V)$ for connected graph
  - Fibonacci heap: $O(E + V \log V)$
  - Array (dense graph): $O(V^2)$
- **Strategy**: Vertex-centric, grows single tree from start
- **Best for**: Dense graphs (many edges)

Borůvka's Algorithm (less common):

- Each component finds its minimum outgoing edge
- Add all such edges simultaneously
- Repeat until one component
- Time: $O(E \log V)$
- Parallelizable

Key Differences - Kruskal vs Prim:

Aspect	Kruskal	Prim
Approach	Edge-based	Vertex-based
Data structure	Union-Find	Priority Queue
Graph type	Sparse graphs	Dense graphs
Start	Any edge	Specific vertex
Growth	Forest → Tree	Single tree
Implementation	Simpler	Slightly complex

Problem-Solving Tips:

- Both produce correct MST (greedy choice property)
- MST weight is unique even if MST structure isn't
- For K_n (complete graph): Choose any algorithm
- If asked for second-best MST: Find MST, then for each MST edge, find best replacement

- GATE trap: Forgetting to check connectivity in Kruskal
- For max spanning tree: Negate weights or reverse comparisons

Verification:

- MST has $V - 1$ edges
- Sum of weights is minimum
- No cycles (DFS/BFS check)
- All vertices reachable (connectivity check)

Examples:

1. Graph: 4 vertices, edges (A,B,1), (B,C,2), (C,D,3), (D,A,4), (A,C,5)
   - Kruskal: Sort → 1,2,3,4,5 → Pick (A,B), (B,C), (C,D) → Weight = 6
   - Prim from A: Pick (A,B,1), then (B,C,2), then (C,D,3) → Weight = 6
2. Finding number of MSTs:
   - If all edges distinct → Unique MST
   - If k edges have same weight and appear in cut → $k!$ possible orderings
3. Updating MST when edge added:
   - If edge connects different MST components → Might reduce total weight
   - Add edge, find cycle, remove max weight edge in cycle

Applications:

- Network design (minimize cable/pipe length)
- Clustering (cut MST edges to form clusters)
- Approximation for TSP (2-approximation via MST)
- Image segmentation
- Taxonomy construction

1.25 Prims Algorithm (2)

(See 1.24)

1.26 Quick Sort (14)

Key Concepts: Partition around pivot; avg $O(nlogn)$, worst $O(n^2)$. **Tips**: Random pivot.

1.27 Recurrence Relation (35)

(See 1.7 for math; here algos like $T(n) = 2T(n/2) + n$).

1.28 Recursion (4)

Key Concepts: Base case, recursive call.

1.29 Searching (7)

(See binary).

1.30 Shortest Path (9)

(See Dijkstra, Bellman).

1.31 Sorting (28)

Key Concepts: Stable, comparison-based $Omega(nlogn)$.

1.32 Space Complexity (1)

Key Concepts: Aux space, e.g., merge $O(n)$.

1.33 Strongly Connected Components (3)

Key Concepts: Kosaraju or Tarjan; two DFS.

1.34 Time Complexity (29)

(See 1.3)

1.35 Topological Sort (4)

Key Concepts: Kahn's indegree 0 queue; $O(V + E)$.

2. CO & Architecture (238)

2.1 Addressing Modes (18)

Key Concepts: Immediate, direct, indirect, indexed. **Tips**: Effective address calc.

2.2 Average Memory Access Time (2)

Key Concepts: Hit time $+$ miss rate $\times$ miss penalty.

2.3 CISC RISC Architecture (2)

Key Concepts: RISC fixed instr, load/store.

2.4 Cache Memory (73)

Key Concepts: Cache memory is a small, fast storage layer between CPU and main memory that exploits locality of reference to improve average memory access time. Critical component in modern computer architecture.

Memory Hierarchy:

```
CPU Registers (1 cycle, ~1KB)
    ↓
L1 Cache (1-2 cycles, 32-64KB)
    ↓
L2 Cache (3-10 cycles, 256KB-1MB)
    ↓
L3 Cache (10-20 cycles, 8-32MB)
    ↓
Main Memory (100-300 cycles, GB)
    ↓
Secondary Storage (10⁶ cycles, TB)
```

Principles of Locality:

1. Temporal Locality: Recently accessed items likely to be accessed again

- Example: Loop variables, frequently called functions
- Exploited by: Keeping recently used data in cache

2. Spatial Locality: Items near recently accessed items likely to be accessed

- Example: Array elements, sequential instruction execution
- Exploited by: Fetching blocks of contiguous data

Cache Organization:

Basic Structure:

- **Cache Line/Block**: Unit of data transfer (typically 32-128 bytes)
- **Tag**: Identifies which memory block is stored
- **Index**: Selects cache set/line
- **Offset**: Byte position within block
- **Valid bit**: Indicates if cache line contains valid data
- **Dirty bit**: Indicates if cache line has been modified (write-back only)

Address Breakdown:

```
|  Tag  | Index | Offset |
|-------|-------|--------|
```

Mapping Techniques:

1. Direct Mapped Cache:

Structure: Each memory block maps to exactly one cache line

- **Index**: `(Block Address) mod (Number of Cache Lines)`

- **Cache Line**: `Index = (Address >> Offset_bits) & Index_mask`

Advantages:

- Simple hardware implementation
- Fast access (no search required)
- Low cost

Disadvantages:

- High conflict miss rate
- Thrashing when accessing alternating blocks that map to same line
- Poor performance for certain access patterns

Example: 1KB cache, 32B blocks, 32-bit addresses

- Cache lines: 1024/32 = 32 lines
- Offset bits: $\log_2(32) = 5$
- Index bits: $\log_2(32) = 5$
- Tag bits: 32 - 5 - 5 = 22

```
struct DirectMappedCache {
    struct CacheLine {
        bool valid;
        int tag;
        char data[BLOCK_SIZE];
    };

    CacheLine lines[NUM_LINES];

    bool access(int address, char& data) {
        int offset = address & OFFSET_MASK;
        int index = (address >> OFFSET_BITS) & INDEX_MASK;
        int tag = address >> (OFFSET_BITS + INDEX_BITS);

        if (lines[index].valid && lines[index].tag == tag) {
            data = lines[index].data[offset];  // Hit
            return true;
        }

        // Miss - fetch from memory
        fetchFromMemory(address, lines[index]);
        lines[index].valid = true;
        lines[index].tag = tag;
        data = lines[index].data[offset];
        return false;
```

```
    }
};
```

2. Fully Associative Cache:

Structure: Any memory block can be placed in any cache line

- **No index bits**: All bits used for tag and offset
- **Search**: Must check all cache lines (parallel comparison)

Advantages:

- Lowest miss rate (no conflict misses)
- Maximum flexibility in placement
- Best utilization of cache space

Disadvantages:

- Expensive hardware (Content Addressable Memory)
- Slower access due to parallel tag comparison
- Complex replacement logic

Implementation:

```
struct FullyAssociativeCache {
    struct CacheLine {
        bool valid;
        int tag;
        int lru_counter;  // For LRU replacement
        char data[BLOCK_SIZE];
    };

    CacheLine lines[NUM_LINES];

    bool access(int address, char& data) {
        int offset = address & OFFSET_MASK;
        int tag = address >> OFFSET_BITS;

        // Search all lines in parallel (hardware)
        for (int i = 0; i < NUM_LINES; i++) {
            if (lines[i].valid && lines[i].tag == tag) {
                data = lines[i].data[offset];  // Hit
                updateLRU(i);
                return true;
            }
        }

        // Miss - find victim using replacement policy
```

```
        int victim = findLRUVictim();
        fetchFromMemory(address, lines[victim]);
        lines[victim].valid = true;
        lines[victim].tag = tag;
        updateLRU(victim);
        data = lines[victim].data[offset];
        return false;
    }
};
```

3. Set Associative Cache (n-way):

Structure: Compromise between direct mapped and fully associative

- **Sets**: Cache divided into sets, each containing n lines
- **Index**: Selects set
- **Tag**: Identifies block within set

Common Configurations: 2-way, 4-way, 8-way, 16-way

Formulas:

- Number of sets = Cache size / (Block size × Associativity)
- Index bits = $\log_2$(Number of sets)
- Tag bits = Address bits - Index bits - Offset bits

Example: 64KB cache, 4-way set associative, 64B blocks

- Sets: 64KB / (64B × 4) = 256 sets
- Index bits: $\log_2(256) = 8$
- Offset bits: $\log_2(64) = 6$
- Tag bits: 32 - 8 - 6 = 18 (for 32-bit addresses)

```
struct SetAssociativeCache {
    struct CacheLine {
        bool valid;
        bool dirty;
        int tag;
        int lru_counter;
        char data[BLOCK_SIZE];
    };

    CacheLine sets[NUM_SETS][ASSOCIATIVITY];

    bool access(int address, char& data) {
        int offset = address & OFFSET_MASK;
        int index = (address >> OFFSET_BITS) & INDEX_MASK;
        int tag = address >> (OFFSET_BITS + INDEX_BITS);
```

```
        // Search within the set
        for (int way = 0; way < ASSOCIATIVITY; way++) {
            if (sets[index][way].valid && sets[index][way].tag == tag) {
                data = sets[index][way].data[offset];  // Hit
                updateLRU(index, way);
                return true;
            }
        }

        // Miss - find victim in this set
        int victim_way = findLRUVictim(index);
        if (sets[index][victim_way].dirty) {
            writeBackToMemory(index, victim_way);
        }

        fetchFromMemory(address, sets[index][victim_way]);
        sets[index][victim_way].valid = true;
        sets[index][victim_way].tag = tag;
        sets[index][victim_way].dirty = false;
        updateLRU(index, victim_way);
        data = sets[index][victim_way].data[offset];
        return false;
    }
};
```

Replacement Policies:

1. Least Recently Used (LRU):

- Replace block that hasn't been used for longest time
- **Implementation**: Counter-based, stack-based, or approximation
- **Hardware cost**: O(n) for n-way associative
- **Performance**: Generally best for typical programs

LRU Implementation (Counter-based):

```
void updateLRU(int set, int way) {
    int current_time = global_counter++;
    sets[set][way].lru_counter = current_time;
}

int findLRUVictim(int set) {
    int lru_way = 0;
    int oldest_time = sets[set][0].lru_counter;

    for (int way = 1; way < ASSOCIATIVITY; way++) {
        if (sets[set][way].lru_counter < oldest_time) {
```

```
            oldest_time = sets[set][way].lru_counter;
            lru_way = way;
        }
    }

    return lru_way;
}
```

2. First In First Out (FIFO):

- Replace oldest block in cache
- **Implementation**: Simple circular buffer
- **Hardware cost**: O(log n) bits per set
- **Performance**: Reasonable, but ignores access patterns

3. Random Replacement:

- Replace randomly selected block
- **Implementation**: Simple random number generator
- **Hardware cost**: Minimal
- **Performance**: Surprisingly good average case

4. Least Frequently Used (LFU):

- Replace block with lowest access frequency
- **Implementation**: Frequency counters
- **Hardware cost**: High (counters for each block)
- **Performance**: Good for some workloads, poor for others

Write Policies:

Write Hit Policies:

1. Write-Through:

- Write to both cache and memory immediately
- **Advantages**: Simple, memory always consistent, no dirty bits needed
- **Disadvantages**: Slower writes, more memory traffic
- **Optimization**: Write buffer to hide memory latency

```
void writeThrough(int address, char data) {
    // Write to cache
    updateCache(address, data);

    // Write to memory (may use write buffer)
    writeBuffer.add(address, data);
```

```
    // Write buffer drains to memory in background
}
```

2. Write-Back (Copy-Back):

- Write only to cache, mark as dirty
- Write to memory only when block is replaced
- **Advantages**: Faster writes, less memory traffic
- **Disadvantages**: Complex, memory inconsistent, needs dirty bits

```
void writeBack(int address, char data) {
    // Write to cache and mark dirty
    updateCache(address, data);
    setCacheLine(address).dirty = true;

    // Memory updated only on eviction
}

void evictLine(CacheLine& line) {
    if (line.dirty) {
        writeToMemory(line.tag, line.data);
    }
    line.valid = false;
    line.dirty = false;
}
```

Write Miss Policies:

1. Write Allocate (Fetch on Write):

- Load block into cache, then write
- **Used with**: Write-back (natural combination)
- **Advantage**: Exploits spatial locality for subsequent accesses

2. No-Write Allocate (Write-No-Allocate):

- Write directly to memory, don't load into cache
- **Used with**: Write-through (natural combination)
- **Advantage**: Avoids polluting cache with write-only data

Performance Analysis:

Key Metrics:

1. Hit Rate (HR): Fraction of accesses that hit in cache

$$HR = \frac{\text{Number of Hits}}{\text{Total Accesses}}$$

2. Miss Rate (MR): Fraction of accesses that miss

$$MR = 1 - HR = \frac{\text{Number of Misses}}{\text{Total Accesses}}$$

3. Average Memory Access Time (AMAT):

$$AMAT = T_{hit} + MR \times T_{miss\_penalty}$$

Where:

- T_{hit} = Cache hit time
- $T_{miss\_penalty}$ = Time to handle cache miss

4. Effective Access Time (EAT):

$$EAT = HR \times T_{cache} + MR \times T_{memory}$$

5. CPI with Cache Misses:

$$CPI_{total} = CPI_{ideal} + \frac{\text{Memory accesses}}{\text{Instruction}} \times MR \times \frac{T_{miss\_penalty}}{T_{clock}}$$

Multi-Level Cache Hierarchy:

Typical Configuration:

- **L1**: Small (32-64KB), fast (1-2 cycles), split I/D
- **L2**: Medium (256KB-1MB), moderate speed (3-10 cycles), unified
- **L3**: Large (8-32MB), slower (10-20 cycles), shared

Miss Rate Types:

- **Local Miss Rate**: Misses in this level / Accesses to this level
- **Global Miss Rate**: Misses in this level / Total CPU memory accesses

Average Access Time for Multi-Level:

$$T_{avg} = T_{L1} + MR_{L1} \times (T_{L2} + MR_{L2} \times T_{memory})$$

For three levels:

$$T_{avg} = T_{L1} + MR_{L1} \times (T_{L2} + MR_{L2} \times (T_{L3} + MR_{L3} \times T_{memory}))$$

Types of Cache Misses (3 C's + 1):

1. Compulsory Misses (Cold Start):

- First access to a block
- **Unavoidable**: Must fetch block at least once

- **Reduction**: Larger block size (limited by miss penalty)

2. Capacity Misses:

- Cache too small to hold working set
- **Would hit in fully associative cache of same size**
- **Reduction**: Larger cache size

3. Conflict Misses (Collision):

- Limited associativity causes eviction
- **Would hit in fully associative cache**
- **Reduction**: Higher associativity

4. Coherence Misses (Multiprocessor):

- Invalidation due to other processors' writes
- **Specific to**: Shared memory multiprocessors
- **Reduction**: Better cache coherence protocols

Cache Optimization Techniques:

1. Larger Block Size:

- **Pros**: Exploits spatial locality, reduces compulsory misses
- **Cons**: Increases miss penalty, may increase conflict misses
- **Optimal**: Balance between spatial locality and miss penalty

2. Higher Associativity:

- **Pros**: Reduces conflict misses
- **Cons**: Increases hit time, more complex hardware
- **Rule of thumb**: 8-way associativity eliminates most conflict misses

3. Larger Cache Size:

- **Pros**: Reduces capacity misses
- **Cons**: Increases cost, may increase hit time
- **Diminishing returns**: Miss rate reduction decreases with size

4. Multi-Level Caches:

- **L1**: Small and fast for common case
- **L2/L3**: Larger to reduce memory accesses
- **Inclusion**: L1 ⊆ L2 ⊆ L3 (common) vs Exclusive

5. Non-Blocking Caches:

- Continue serving hits while handling misses
- **Hit under miss**: Serve cache hits during miss processing
- **Miss under miss**: Handle multiple outstanding misses

6. Prefetching:

- **Hardware prefetching**: Detect patterns, fetch ahead
- **Software prefetching**: Compiler/programmer inserts prefetch instructions
- **Stream buffers**: Detect sequential access patterns

Cache-Conscious Programming:

1. Spatial Locality:

```
// Good: Access array elements sequentially
for (int i = 0; i < N; i++) {
    sum += array[i];
}

// Bad: Large stride access
for (int i = 0; i < N; i += 1000) {
    sum += array[i];
}
```

2. Temporal Locality:

```
// Good: Reuse data while in cache
for (int i = 0; i < N; i++) {
    temp = array[i];
    result1 += temp * temp;
    result2 += temp + 1;
}

// Bad: Multiple passes over same data
for (int i = 0; i < N; i++) result1 += array[i] * array[i];
for (int i = 0; i < N; i++) result2 += array[i] + 1;
```

3. Cache Blocking (Tiling):

```
// Matrix multiplication with blocking
for (int ii = 0; ii < N; ii += BLOCK_SIZE) {
    for (int jj = 0; jj < N; jj += BLOCK_SIZE) {
        for (int kk = 0; kk < N; kk += BLOCK_SIZE) {
            // Work on BLOCK_SIZE x BLOCK_SIZE submatrices
            for (int i = ii; i < min(ii + BLOCK_SIZE, N); i++) {
                for (int j = jj; j < min(jj + BLOCK_SIZE, N); j++) {
                    for (int k = kk; k < min(kk + BLOCK_SIZE, N); k++) {
```

```
                        C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
    }
}
```

Problem-Solving Strategy:

1. Identify Cache Parameters:

- Cache size, block size, associativity
- Calculate: sets, index bits, offset bits, tag bits

2. Trace Memory Accesses:

- For each access: extract tag, index, offset
- Check for hit/miss in appropriate set
- Apply replacement policy on miss

3. Calculate Performance:

- Count hits and misses
- Compute hit rate, miss rate
- Calculate AMAT using given timing parameters

GATE Tips:

- **Address breakdown**: Tag | Index | Offset
- **Block size**: Always power of 2, determines offset bits
- **Direct mapped**: Index = (Block address) mod (Cache lines)
- **Set associative**: Sets = Cache size / (Block size × Ways)
- **Fully associative**: No index bits, all tag + offset
- **LRU**: Best replacement policy for most workloads
- **Write-back**: Faster but needs dirty bits
- **Write-through**: Simpler but slower
- **AMAT**: Hit time + Miss rate × Miss penalty
- **Multi-level**: Local vs global miss rates
- **3 C's**: Compulsory, Capacity, Conflict (+ Coherence)

Common GATE Problems:

Example 1: 16KB direct-mapped cache, 32B blocks, 32-bit addresses

- Cache lines: 16KB / 32B = 512 lines

- Offset bits: $\log_2(32) = 5$
- Index bits: $\log_2(512) = 9$
- Tag bits: 32 - 5 - 9 = 18

Example 2: Hit rate 90%, cache time 2ns, memory time 50ns

- AMAT = 2 + 0.1 × 50 = 7ns

Example 3: 2-level cache, L1 hit rate 95%, L2 hit rate 80%

- L1 time: 1ns, L2 time: 5ns, Memory time: 100ns
- AMAT = 1 + 0.05 × (5 + 0.2 × 100) = 1 + 0.05 × 25 = 2.25ns

Example 4: Cache access sequence analysis

- Given: 4-way set associative, 8 sets, LRU replacement
- Trace through access sequence, track hits/misses
- Apply LRU policy for each set independently

2.5 Clock Cycles (1)

Key Concepts: CPI $= fractextcyclestextinstr.$

2.6 DMA (9)

Key Concepts: Direct mem access bypassing CPU.

2.7 Data Dependency (2)

Key Concepts: RAW, WAR, WAW; true/false.

2.8 Data Path (7)

Key Concepts: ALU, registers, mux.

2.9 IO Handling (7)

Key Concepts: Polling vs interrupt.

2.10 Instruction Execution (6)

Key Concepts: Fetch-decode-execute.

2.11 Instruction Format (9)

Key Concepts: Opcode, operands bits.

2.12 Instruction Set Architecture (1)

Key Concepts: ISA levels.

2.13 Interrupts (9)

Key Concepts: Vectored, priority.

2.14 Machine Instruction (21)

Key Concepts: Types: arithmetic, control.

2.15 Memory Interfacing (6)

Key Concepts: Address decoding.

2.16 Microprogramming (13)

Key Concepts: Firmware control store.

2.17 Pipelining (43)

Key Concepts: Pipelining is a technique that overlaps execution of multiple instructions to improve processor throughput. It divides instruction execution into stages that can operate concurrently.

Basic Pipeline Concept:

Sequential Execution: Instructions execute one after another

- Time per instruction = Sum of all stage delays
- Throughput = 1 instruction per total execution time

Pipelined Execution: Instructions overlap in different stages

- Time per instruction ≈ Time of slowest stage (ideally)
- Throughput = 1 instruction per clock cycle (ideally)

Classic 5-Stage RISC Pipeline:

1. IF (Instruction Fetch):

- Fetch instruction from memory using PC
- Update PC to next instruction
- Send instruction to next stage

2. ID (Instruction Decode):

- Decode instruction opcode and fields
- Read source registers from register file
- Generate control signals

- Compute branch target address

3. EX (Execute):

- Perform ALU operations
- Calculate memory addresses
- Evaluate branch conditions
- Forward results if needed

4. MEM (Memory Access):

- Access data memory for loads/stores
- Write data to memory (stores)
- Read data from memory (loads)
- Pass through other instructions

5. WB (Write Back):

- Write results back to register file
- Update destination registers
- Complete instruction execution

Pipeline Registers: Store intermediate results between stages

- **IF/ID**: Instruction, PC+4
- **ID/EX**: Control signals, register values, immediate values
- **EX/MEM**: ALU result, memory data, control signals
- **MEM/WB**: Memory data or ALU result, destination register

Performance Analysis:

Ideal Speedup: $S = \frac{\text{Time without pipelining}}{\text{Time with pipelining}}$

For k-stage pipeline with n instructions:

- **Without pipelining**: $n \times k \times t$ (where t = time per stage)
- **With pipelining**: $(k + n - 1) \times t$
- **Speedup**: $S = \frac{n \times k}{k + n - 1}$

For large n: $S \approx k$ (number of pipeline stages)

Throughput: Instructions completed per unit time

- **Ideal**: 1 instruction per clock cycle
- **Actual**: Reduced by hazards and stalls

CPI (Cycles Per Instruction):

- **Ideal pipelined CPI**: 1.0
- **Actual CPI**: $CPI_{ideal} + CPI_{stalls}$
- **Overall CPI**: $CPI = 1 + \text{Stall cycles per instruction}$

Pipeline Hazards:

1. Structural Hazards:
Cause: Hardware resource conflicts
Example: Single memory for both instructions and data

Solutions:

- **Separate memories**: Harvard architecture (separate I-cache and D-cache)
- **Resource duplication**: Multiple ALUs, register file ports
- **Pipeline stalls**: Wait for resource availability

2. Data Hazards:
Cause: Instructions depend on results of previous instructions

Types of Data Dependencies:

RAW (Read After Write) - True Dependency:

```
add r1, r2, r3    # r1 = r2 + r3
sub r4, r1, r5    # r4 = r1 - r5 (needs r1 from add)
```

- Most common and problematic
- Later instruction needs result from earlier instruction

WAR (Write After Read) - Anti-dependency:

```
add r1, r2, r3    # reads r2, r3
sub r2, r4, r5    # writes r2
```

- Rare in simple pipelines (in-order execution)
- Can occur with out-of-order execution

WAW (Write After Write) - Output Dependency:

```
add r1, r2, r3    # writes r1
sub r1, r4, r5    # also writes r1
```

- Rare in simple pipelines
- Can occur with out-of-order execution or multiple functional units

Data Hazard Solutions:

A. Pipeline Stalls (Bubbles):

- Insert NOPs until data is available
- **Disadvantage**: Reduces performance
- **Implementation**: Hazard detection unit stalls pipeline

```
Cycle:  1  2  3  4  5  6  7  8  9
add:   IF ID EX MEM WB
sub:      IF ID -- -- EX MEM WB    (2 stall cycles)
```

B. Data Forwarding (Bypassing):

- Forward results directly from later pipeline stages
- **Advantage**: Eliminates many stalls
- **Implementation**: Forwarding unit with multiplexers

Forwarding Paths:

- **EX/MEM → EX**: ALU result to ALU input
- **MEM/WB → EX**: Memory data or ALU result to ALU input
- **MEM/WB → ID**: For register file write-back

```
// Forwarding logic
if (EX_MEM.RegWrite && EX_MEM.RegisterRd == ID_EX.RegisterRs) {
    ForwardA = 10;  // Forward from EX/MEM
}
else if (MEM_WB.RegWrite && MEM_WB.RegisterRd == ID_EX.RegisterRs) {
    ForwardA = 01;  // Forward from MEM/WB
}
else {
    ForwardA = 00;  // No forwarding
}
```

Load-Use Hazard: Special case requiring stall

```
lw  r1, 0(r2)     # Load r1 from memory
add r3, r1, r4    # Use r1 immediately
```

- Data not available until MEM stage
- Forwarding cannot eliminate this stall
- **Solution**: 1 cycle stall + forwarding

3. Control Hazards (Branch Hazards):
Cause: Uncertainty about next instruction due to branches

Branch Penalty: Cycles lost due to incorrect speculation

- **Worst case**: Flush entire pipeline
- **Typical**: 1-3 cycle penalty

Control Hazard Solutions:

A. Pipeline Stalls:

- Stall until branch outcome known
- **Penalty**: Number of stages before branch resolution
- Simple but inefficient

B. Branch Prediction:

Static Prediction:

- **Always taken**: Assume all branches taken
- **Always not taken**: Assume all branches not taken
- **Backward taken, forward not taken**: Based on branch direction

Dynamic Prediction:

1-bit Predictor: Remember last branch outcome

- **Problem**: Alternating pattern causes 100% misprediction

2-bit Predictor (Saturating Counter):

```
States: 00 (Strongly Not Taken)
        01 (Weakly Not Taken)
        10 (Weakly Taken)
        11 (Strongly Taken)

Transitions:
- Taken: increment (saturate at 11)
- Not taken: decrement (saturate at 00)
- Predict taken if state ≥ 10
```

Branch Target Buffer (BTB):

- Cache of branch instructions and their targets
- Enables prediction of both direction and target
- **Structure**: PC → (Target Address, Prediction)

C. Delayed Branches:

- Execute instruction after branch regardless of outcome

- **Delay slot**: Instruction position after branch
- Compiler fills with useful instruction or NOP
- **Example**: MIPS architecture

```
beq r1, r2, target
add r3, r4, r5      # Delay slot - always executed
target: ...
```

D. Branch Prediction with Speculation:

- Predict branch outcome and continue fetching
- **Correct prediction**: No penalty
- **Incorrect prediction**: Flush pipeline, restart from correct path

Advanced Pipeline Concepts:

Superscalar Pipelines:

- Multiple instructions issued per cycle
- **Challenges**: More complex hazard detection, resource conflicts
- **Example**: Issue 2-4 instructions per cycle

Out-of-Order Execution:

- Instructions execute when operands ready, not in program order
- **Advantages**: Better resource utilization, hide latencies
- **Challenges**: WAR and WAW hazards, complex control

Pipeline Depth Trade-offs:

Deeper Pipelines (more stages):

- **Advantages**: Higher clock frequency, better throughput
- **Disadvantages**: Higher branch penalty, more complex hazard handling
- **Example**: Pentium 4 had 20+ stage pipeline

Shallower Pipelines (fewer stages):

- **Advantages**: Lower branch penalty, simpler design
- **Disadvantages**: Lower maximum clock frequency
- **Example**: ARM Cortex-A8 has 13-stage pipeline

Pipeline Performance Metrics:

Speedup Calculation:
$$\text{Speedup} = \frac{\text{Execution time without pipelining}}{\text{Execution time with pipelining}}$$

For n instructions, k-stage pipeline:
$\text{Speedup} = \frac{n \times k}{k+n-1+\text{stall cycles}}$

CPI with Hazards:
$CPI = CPI_{ideal} + CPI_{structural} + CPI_{data} + CPI_{control}$

Where:

- $CPI_{ideal} = 1.0$ (for simple pipeline)
- $CPI_{structural}$ = stalls due to resource conflicts
- CPI_{data} = stalls due to data hazards
- $CPI_{control}$ = stalls due to control hazards

Example Calculation:
Given:

- 20% branches, 3-cycle branch penalty
- 30% loads, 1-cycle load-use penalty (25% of loads)
- Perfect branch prediction 85% of time

$CPI_{control} = 0.20 \times 0.15 \times 3 = 0.09$
$CPI_{data} = 0.30 \times 0.25 \times 1 = 0.075$
$CPI_{total} = 1.0 + 0.09 + 0.075 = 1.165$

Pipeline Implementation Details:

Hazard Detection Unit:

```
class HazardDetectionUnit {
public:
    bool detectLoadUseHazard(int ID_EX_rt, int IF_ID_rs, int IF_ID_rt) {
        return (ID_EX_MemRead &&
                ((ID_EX_rt == IF_ID_rs) || (ID_EX_rt == IF_ID_rt)));
    }

    bool detectRAWHazard(int EX_MEM_rd, int ID_EX_rs, int ID_EX_rt) {
        return (EX_MEM_RegWrite && EX_MEM_rd != 0 &&
                ((EX_MEM_rd == ID_EX_rs) || (EX_MEM_rd == ID_EX_rt)));
    }
};
```

Forwarding Unit:

```
class ForwardingUnit {
public:
    int getForwardA(int ID_EX_rs, int EX_MEM_rd, int MEM_WB_rd) {
        if (EX_MEM_RegWrite && EX_MEM_rd != 0 && EX_MEM_rd == ID_EX_rs)
            return 2;  // Forward from EX/MEM
        else if (MEM_WB_RegWrite && MEM_WB_rd != 0 && MEM_WB_rd == ID_EX_rs)
```

```
            return 1;  // Forward from MEM/WB
        else
            return 0;  // No forwarding
    }
};
```

Branch Predictor:

```
class BranchPredictor {
    vector<int> predictorTable;  // 2-bit saturating counters

public:
    bool predict(int pc) {
        int index = (pc >> 2) & (predictorTable.size() - 1);
        return predictorTable[index] >= 2;  // Predict taken if ≥ 2
    }

    void update(int pc, bool taken) {
        int index = (pc >> 2) & (predictorTable.size() - 1);
        if (taken && predictorTable[index] < 3)
            predictorTable[index]++;
        else if (!taken && predictorTable[index] > 0)
            predictorTable[index]--;
    }
};
```

Problem-Solving Strategy:

1. Identify Pipeline Structure:

- Number of stages
- Function of each stage
- Pipeline registers between stages

2. Trace Instruction Execution:

- Show which stage each instruction is in each cycle
- Identify when hazards occur
- Apply hazard resolution techniques

3. Calculate Performance:

- Count total cycles including stalls
- Compute CPI and speedup
- Consider impact of different hazard types

GATE Tips:

- **5-stage pipeline**: IF-ID-EX-MEM-WB is standard
- **RAW hazards**: Most common, solved by forwarding
- **Load-use hazard**: Always requires 1 stall cycle
- **Branch penalty**: Depends on when branch resolved
- **Forwarding paths**: EX/MEM→EX, MEM/WB→EX most important
- **CPI calculation**: 1 + stall cycles per instruction
- **Speedup**: Approaches number of stages for large programs
- **2-bit predictor**: Better than 1-bit for loops
- **Delayed branch**: Instruction after branch always executes
- **Pipeline depth**: Trade-off between frequency and branch penalty

Common GATE Problems:

Example 1: Pipeline timing diagram

- Show 5 instructions through 5-stage pipeline
- Identify hazards and show stalls/forwarding

Example 2: CPI calculation

- Given: 25% branches (2-cycle penalty), 20% loads (1-cycle stall for 30%)
- CPI = 1 + 0.25×2 + 0.20×0.30×1 = 1.56

Example 3: Speedup calculation

- 1000 instructions, 5-stage pipeline, 100 stall cycles
- Without pipelining: 1000×5 = 5000 cycles
- With pipelining: 5+1000-1+100 = 1104 cycles
- Speedup = 5000/1104 = 4.53

Example 4: Forwarding analysis

```
add r1, r2, r3    # Cycle 1-5
sub r4, r1, r5    # Cycle 2-6, needs r1 from add
and r6, r1, r7    # Cycle 3-7, needs r1 from add
```

- Forward from EX/MEM to EX for sub instruction
- Forward from MEM/WB to EX for and instruction
- No stalls needed with proper forwarding

2.18 Runtime Environment (2)

Key Concepts: Stack frames.

2.19 Speedup (4)

(See 2.17)

2.20 Virtual Memory (3)

Key Concepts: Paging, TLB.

3. Compiler Design (234)

3.1 Assembler (9)

Key Concepts: Two-pass, symbol table.

3.2 Backpatching (1)

Key Concepts: Fix forward refs in code gen.

3.3 Basic Blocks (1)

Key Concepts: Sequential code no branches.

3.4 Code Optimization (10)

Key Concepts: Local/global, peephole.

3.5 Compilation Phases (12)

Key Concepts: Lex -> parse -> semantic -> code gen -> opt.

3.6 Expression Evaluation (2)

Key Concepts: Postfix, operator precedence.

3.7 First and Follow (6)

Key Concepts: For LL parsing; FIRST(A) non-ε starters.

3.8 Grammar (51)

Key Concepts: Grammars are formal systems for defining the syntax of programming languages and natural languages. They specify the structure and rules for generating valid strings in a language.

Formal Grammar Definition:

A grammar G is a 4-tuple: **G = (V, T, P, S)** where:

- **V**: Finite set of **non-terminals** (variables, syntactic categories)
- **T**: Finite set of **terminals** (alphabet, tokens)
- **P**: Finite set of **productions** (rewrite rules)
- **S**: **Start symbol** ($S \in V$, root of derivation)

Constraints: $V \cap T = \emptyset$ (disjoint sets)

Production Rules: $A \rightarrow \alpha$ where:

- $A \in V$ (left-hand side is non-terminal)
- $\alpha \in (V \cup T)^*$ (right-hand side is string of symbols)

Example Grammar:

```
G = ({S, A}, {a, b}, P, S)
P: S → aAb
   A → aAb | ε
```

Generates: $\{a^n b^n \mid n \geq 1\}$

Derivations and Parse Trees:

Derivation: Sequence of production applications to generate strings

- **Direct derivation**: $\alpha \Rightarrow \beta$ (one production application)
- **Derivation**: $\alpha \Rightarrow^* \beta$ (zero or more steps)
- **Language**: $L(G) = \{w \in T \mid S \Rightarrow w\}$

Types of Derivations:

1. Leftmost Derivation: Always expand leftmost non-terminal first

```
S ⇒ aAb ⇒ aaAbb ⇒ aabb
```

2. Rightmost Derivation: Always expand rightmost non-terminal first

```
S ⇒ aAb ⇒ aaAbb ⇒ aabb
```

3. Parse Tree: Hierarchical representation of derivation

- **Root**: Start symbol
- **Internal nodes**: Non-terminals
- **Leaves**: Terminals (left-to-right gives derived string)
- **Unique**: Same parse tree for all derivation orders

Sentential Forms:

- **Sentential form**: Any string derivable from start symbol
- **Sentence**: Sentential form containing only terminals
- **Left-sentential form**: Derived by leftmost derivation
- **Right-sentential form**: Derived by rightmost derivation

Ambiguity in Grammars:

Definition: Grammar is **ambiguous** if ∃ string with:

- Multiple distinct parse trees, OR
- Multiple leftmost derivations, OR
- Multiple rightmost derivations

Classic Example - Arithmetic Expressions:

```
Ambiguous Grammar:
E → E + E | E * E | (E) | id

String: id + id * id

Parse Tree 1:     E           Parse Tree 2:     E
                / \ \                         / | \
               E   *  E                      E  +  E
              /|\     |                     |    /|\
             E + E   id                    id   E * E
             |   |                             |   |
            id  id                           id  id

Meaning 1: (id + id) * id    Meaning 2: id + (id * id)
```

Resolving Ambiguity:

Method 1: Operator Precedence

```
Unambiguous Grammar:
E → E + T | T          // Addition (lower precedence)
T → T * F | F          // Multiplication (higher precedence)
F → (E) | id           // Factors (highest precedence)

Now "id + id * id" uniquely parses as "id + (id * id)"
```

Method 2: Operator Associativity

Left Associative (most binary operators):

```
E → E + T | T          // Left recursion
// a + b + c = ((a + b) + c)
```

Right Associative (assignment, exponentiation):

```
E → T ^ E | T          // Right recursion
// a ^ b ^ c = (a ^ (b ^ c))
```

```
A → id = A | id        // Assignment
// a = b = c means a = (b = c)
```

Method 3: Dangling Else Resolution

Problem:

```
S → if E then S | if E then S else S | other

Ambiguous string: if E1 then if E2 then S1 else S2
```

Solution: Else matches closest unmatched if

```
S → M | U                    // Matched and Unmatched statements
M → if E then M else M       // Both branches present
  | other
U → if E then S              // Missing else branch
  | if E then M else U       // Unmatched in else branch
```

Chomsky Hierarchy:

Type 0: Unrestricted Grammar (Phrase Structure)

- **Productions**: $\alpha \rightarrow \beta$ where $\alpha \in (V \cup T)^+$, $\beta \in (V \cup T)^*$
- **Restriction**: α must contain at least one non-terminal
- **Languages**: Recursively Enumerable (RE)
- **Automaton**: Turing Machine
- **Example**: Context-sensitive features, natural language

Type 1: Context-Sensitive Grammar (CSG)

- **Productions**: $\alpha A \beta \rightarrow \alpha \gamma \beta$ where $A \in V$, $\alpha,\beta,\gamma \in (V \cup T)^*$, $\gamma \neq \varepsilon$
- **Length constraint**: $|\alpha| \leq |\beta|$ (non-contracting)
- **Exception**: $S \rightarrow \varepsilon$ allowed if S doesn't appear on RHS
- **Languages**: Context-Sensitive Languages (CSL)
- **Automaton**: Linear Bounded Automaton
- **Example**: $\{a^n b^n c^n \mid n \geq 1\}$

Type 2: Context-Free Grammar (CFG)

- **Productions**: $A \rightarrow \alpha$ where $A \in V$, $\alpha \in (V \cup T)^*$
- **Restriction**: LHS must be single non-terminal
- **Languages**: Context-Free Languages (CFL)
- **Automaton**: Pushdown Automaton

- **Example**: Programming language syntax, balanced parentheses

Type 3: Regular Grammar

- **Right-linear**: A → wB | w where A,B ∈ V, w ∈ T*
- **Left-linear**: A → Bw | w where A,B ∈ V, w ∈ T*
- **Cannot mix**: Left and right linear in same grammar
- **Languages**: Regular Languages
- **Automaton**: Finite Automaton
- **Example**: Lexical tokens, simple patterns

Language Hierarchy: Regular ⊂ Context-Free ⊂ Context-Sensitive ⊂ Recursively Enumerable

Normal Forms:

Chomsky Normal Form (CNF):

Definition: All productions have form:

- A → BC (two non-terminals)
- A → a (single terminal)
- S → ε (only if ε ∈ L(G) and S doesn't appear on RHS)

Uses:

- CYK parsing algorithm O(n³)
- Proving pumping lemma for CFLs
- Theoretical analysis

Conversion Algorithm:

1. **Eliminate ε-productions**: Remove A → ε
2. **Eliminate unit productions**: Remove A → B
3. **Replace terminals**: In productions with length > 1
4. **Break long productions**: Into binary form

Example Conversion:

```
Original: S → ASA | aB, A → B | S, B → b | ε

Step 1 (Remove ε):
S → ASA | aB | a
A → B | S | ε
B → b

Step 2 (Remove unit productions):
```

```
S → ASA | aB | a | b | ASA
A → b | ASA | aB | a
B → b

Step 3 (Replace terminals):
S → ASA | XB | a | b
A → b | ASA | XB | a
B → b
X → a

Step 4 (Break long productions):
S → AY | XB | a | b
A → b | AY | XB | a
B → b
X → a
Y → SA
```

Greibach Normal Form (GNF):

Definition: All productions have form:

- $A \rightarrow a\alpha$ where $a \in T$, $\alpha \in V^*$
- $S \rightarrow \varepsilon$ (only if $\varepsilon \in L(G)$)

Properties:

- Every derivation step increases string length
- Useful for eliminating left recursion
- Enables construction of PDA without ε-moves

Grammar Transformations:

Eliminating Useless Symbols:

Non-generating symbols: Cannot derive terminal strings

```
Algorithm:
1. Mark all terminals as generating
2. Mark A as generating if A → α and all symbols in α are generating
3. Repeat until no new symbols marked
4. Remove unmarked symbols
```

Unreachable symbols: Cannot be reached from start symbol

```
Algorithm:
1. Mark start symbol as reachable
2. If A is reachable and A → α, mark all symbols in α as reachable
```

```
3. Repeat until no new symbols marked
4. Remove unmarked symbols
```

Eliminating ε-Productions:

Nullable symbols: Can derive ε

```
Algorithm:
1. Mark A as nullable if A → ε
2. Mark A as nullable if A → α and all symbols in α are nullable
3. For each production A → α with nullable symbols:
   - Add all combinations with nullable symbols present/absent
4. Remove explicit ε-productions (except S → ε if needed)
```

Example:

```
Original: S → AB, A → a | ε, B → b | ε
Nullable: A, B

New productions:
S → AB | A | B | ε
A → a
B → b
```

Eliminating Unit Productions:

Unit production: A → B where A,B ∈ V

```
Algorithm:
1. Find all unit pairs (A,B) where A ⇒* B via unit productions
2. For each unit pair (A,B) and non-unit production B → α:
   - Add production A → α
3. Remove all unit productions
```

Left Recursion Elimination:

Immediate Left Recursion: A → Aα | β

Transformation:

```
A → Aα₁ | Aα₂ | ... | Aαₘ | β₁ | β₂ | ... | βₙ

Becomes:
A → β₁A' | β₂A' | ... | βₙA'
A' → α₁A' | α₂A' | ... | αₘA' | ε
```

Example:

```
E → E + T | E − T | T

Becomes:
E → TE'
E' → +TE' | −TE' | ε
```

Indirect Left Recursion: $A \Rightarrow^{+} A\alpha$

Algorithm:

1. Order non-terminals: $A_1, A_2, ..., A_n$
2. For i = 1 to n:
   - For j = 1 to i-1:
     - Replace $A_i \rightarrow A_j\alpha$ with $A_i \rightarrow \beta_1\alpha \mid \beta_2\alpha \mid ...$ where $A_j \rightarrow \beta_1 \mid \beta_2 \mid ...$
   - Eliminate immediate left recursion from A_i

Left Factoring:

Problem: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid ...$

Solution:

```
A → αA' | γ₁ | γ₂ | ...
A' → β₁ | β₂ | ... | βₙ
```

Context-Free Language Properties:

Closure Properties (CFLs are closed under):

- **Union**: $L_1 \cup L_2$
- **Concatenation**: $L_1 \cdot L_2$
- **Kleene Star**: L^*
- **Substitution**: Replace each symbol with a language
- **Homomorphism**: $h(L) = \{h(w) \mid w \in L\}$
- **Inverse Homomorphism**: $h^{-1}(L) = \{w \mid h(w) \in L\}$
- **Reversal**: $L^R = \{w^R \mid w \in L\}$

Non-Closure Properties (CFLs are NOT closed under):

- **Intersection**: $\{a^n b^n c^m\} \cap \{a^m b^n c^n\} = \{a^n b^n c^n\}$ (not CFL)
- **Complement**: Due to non-closure under intersection

Special Case: CFL ∩ Regular = CFL (intersection with regular language)

Pumping Lemma for Context-Free Languages:

Statement: For every CFL L, ∃ pumping length p such that:
Every string w ∈ L with |w| ≥ p can be written as w = uvxyz where:

1. |vxy| ≤ p
2. |vy| ≥ 1 (at least one of v,y is non-empty)
3. ∀i ≥ 0: $uv^ixy^iz \in L$

Intuition: Long strings must have repeated non-terminals in parse tree

Proof Technique:

1. Assume L is CFL
2. Choose w strategically (often w = $a^pb^pc^p$)
3. Apply pumping lemma decomposition
4. Show pumping violates language definition
5. Conclude L is not CFL

Example: L = {$a^nb^nc^n$ | n ≥ 0} is not CFL

- Choose w = $a^pb^pc^p$
- By |vxy| ≤ p, vxy spans at most 2 symbol types
- Pumping changes symbol ratios
- $uv^ixy^iz \notin L$ for some i ≠ 1

Decision Problems:

Membership Problem: Given CFG G and string w, is w ∈ L(G)?

- **Algorithm**: CYK algorithm
- **Complexity**: $O(n^3)$ for CNF grammar
- **Decidable**: Yes

Emptiness Problem: Given CFG G, is L(G) = ∅?

- **Algorithm**: Check if start symbol is generating
- **Complexity**: O(|G|)
- **Decidable**: Yes

Finiteness Problem: Given CFG G, is L(G) finite?

- **Algorithm**: Check for cycles in dependency graph
- **Complexity**: O(|G|)
- **Decidable**: Yes

Equivalence Problem: Given CFGs G_1, G_2, is $L(G_1) = L(G_2)$?

- **Decidable**: No (undecidable)

Inherent Ambiguity: Some CFLs have no unambiguous grammar

- **Example**: $\{a^i b^j c^k \mid i=j \text{ or } j=k\}$
- **Proof**: Any grammar must have ambiguous derivations

Applications in Compiler Design:

Syntax Analysis:

- **Top-down parsing**: LL grammars
- **Bottom-up parsing**: LR grammars
- **Parse tree construction**: AST generation

Language Design:

- **Operator precedence**: Expression grammars
- **Statement structure**: Control flow
- **Type declarations**: Variable definitions

Error Recovery:

- **Panic mode**: Skip to synchronizing token
- **Phrase level**: Local corrections
- **Error productions**: Anticipate common errors

Problem-Solving Strategy:

Grammar Analysis:

1. Identify grammar type (regular, context-free, etc.)
2. Check for ambiguity (find multiple parse trees)
3. Apply transformations if needed (eliminate left recursion, left factor)
4. Verify language properties (closure, decidability)

Language Proof:

1. Show $L(G) \subseteq$ intended language (every generated string is valid)
2. Show intended language $\subseteq L(G)$ (every valid string can be generated)
3. Use structural induction on derivations

GATE Tips:

- **Ambiguity**: Multiple parse trees for same string
- **Left recursion**: Incompatible with LL parsing
- **CNF**: Useful for CYK algorithm, complexity analysis

- **Pumping lemma**: Prove languages are not context-free
- **Closure**: CFLs closed under union, concatenation, star
- **Non-closure**: CFLs not closed under intersection, complement
- **Hierarchy**: Regular ⊂ CFL ⊂ CSL ⊂ RE
- **Decidability**: Membership, emptiness decidable; equivalence undecidable
- **Normal forms**: CNF for parsing, GNF for PDA construction
- **Transformations**: Essential for parser construction

Common GATE Examples:

Example 1: Show grammar is ambiguous

```
S → S + S | S * S | (S) | a
String: a + a * a

Two parse trees possible:
1. (a + a) * a
2. a + (a * a)
```

Example 2: Eliminate left recursion

```
A → Aa | Ab | c | d

Becomes:
A → cA' | dA'
A' → aA' | bA' | ε
```

Example 3: Convert to CNF

```
S → ASA | aB
A → B | S
B → b | ε

After conversion:
S → AY | XB | a | b
A → b | AY | XB | a
B → b
X → a
Y → SA
```

Example 4: Use pumping lemma

```
Prove L = {aⁿbⁿcⁿ | n ≥ 0} is not CFL:
- Choose w = aᵖbᵖcᵖ
- vxy spans ≤ 2 symbol types
```

```
- Pumping destroys balance
- Therefore not CFL
```

3.9 Intermediate Code (11)

Key Concepts: Three-address: t1 = a + b.

3.10 LR Parser (20)

Key Concepts: SLR/LALR, shift-reduce.

3.11 Lexical Analysis (6)

Key Concepts: Tokens, regex to NFA.

3.12 Linker (3)

Key Concepts: Resolve symbols, static/dynamic.

3.13 Live Variable Analysis (3)

Key Concepts: Backward dataflow.

3.14 Macros (4)

Key Concepts: Expansion, parameters.

3.15 Operator Precedence (3)

Key Concepts: Grammar for expr.

3.16 Parameter Passing (14)

Key Concepts: Call-by-value/ref.

3.17 Parsing (27)

Key Concepts: Top-down/bottom-up.

3.18 Register Allocation (6)

Key Concepts: Graph coloring.

3.19 Runtime Environment (22)

(See 2.18)

3.20 Static Single Assignment (3)

Key Concepts: φ functions for branches.

3.21 Symbol Table (1)

Key Concepts: Hash for scopes.

3.22 Syntax Directed Translation (17)

Key Concepts: Attributes, S-attrib/L-attrib.

3.23 Variable Scope (2)

Key Concepts: Static/dynamic binding.

4. Computer Networks (215)

4.1 Application Layer Protocols (12)

Key Concepts: HTTP stateless, FTP control/data.

4.2 Arp (1)

Key Concepts: IP to MAC mapping.

4.3 Bit Stuffing (2)

Key Concepts: 0 after 5 ones in HDLC.

4.4 Bridges (3)

Key Concepts: L2 forwarding, learning.

4.5 CRC Polynomial (4)

Key Concepts: Mod 2 division for error detect.

4.6 CSMA CD (6)

Key Concepts: Ethernet collision detect; slot time 2τ.

4.7 Channel Utilization (1)

Key Concepts: Efficiency $S = 1/(1+6a+\ldots)$.

4.8 Communication (4)

Key Concepts: Simplex/half/full duplex.

4.9 Congestion Control (6)

Key Concepts: TCP slow start, AIMD.

4.10 Distance Vector Routing (8)

Key Concepts: Bellman-Ford distributed; count-to-inf.

4.11 Error Detection (10)

(See CRC)

4.12 Ethernet (7)

Key Concepts: CSMA/CD, 64B min frame.

4.13 Fragmentation (2)

Key Concepts: IP MTU, offset*8.

4.14 IP Addressing (11)

Key Concepts: CIDR /n, subnet mask.

4.15 IP Packet (11)

Key Concepts: Header checksum, TTL.

4.16 LAN Technologies (7)

Key Concepts: Ethernet speeds.

4.17 MAC Protocol (5)

(See 4.6)

4.18 Network Flow (4)

Key Concepts: Max flow min cut.

4.19 Network Layering (6)

Key Concepts: OSI 7 layers.

4.20 Network Protocols (10)

(See specific)

4.21 Network Switching (4)

Key Concepts: Circuit/packet.

4.22 Osi Model (1)

(See 4.19)

4.23 Probability (1)

(See Vol1 prob)

4.24 Routing (13)

(See 4.10)

4.25 Routing Protocols (1)

Key Concepts: RIP, OSPF link-state.

4.26 Sliding Window (15)

Key Concepts: GBN/SRW; window size N.

4.27 Sockets (4)

Key Concepts: API bind/listen/accept.

4.28 Stop and Wait (6)

Key Concepts: Efficiency 1/(1+2a).

4.29 Subnetting (21)

(See 4.14)

4.30 TCP (22)

Key Concepts: Transmission Control Protocol - reliable, connection-oriented, byte-stream service. Ensures ordered, error-free delivery.

TCP Features:

- **Connection-oriented**: Establishes connection before data transfer
- **Reliable**: Guarantees delivery via ACKs and retransmission
- **Ordered**: Maintains sequence numbers
- **Flow control**: Prevents sender from overwhelming receiver
- **Congestion control**: Prevents network congestion
- **Full-duplex**: Bidirectional communication
- **Byte-stream**: No message boundaries (unlike UDP)

TCP Segment Structure:

- **Source/Destination Port**: 16 bits each (identifies applications)
- **Sequence Number**: 32 bits (byte number of first data byte)
- **Acknowledgment Number**: 32 bits (next expected byte)
- **Header Length**: 4 bits (in 32-bit words, typically 5)
- **Flags**: 6 bits

    - **URG**: Urgent pointer valid
    - **ACK**: Acknowledgment number valid
    - **PSH**: Push data to application
    - **RST**: Reset connection
    - **SYN**: Synchronize sequence numbers (connection setup)
    - **FIN**: Finish, no more data (connection teardown)
- **Window Size**: 16 bits (for flow control, in bytes)
- **Checksum**: 16 bits (error detection)
- **Urgent Pointer**: 16 bits (if URG set)
- **Options**: Variable (e.g., MSS, timestamps, window scaling)
- **Data**: Variable length

Three-Way Handshake (Connection Establishment):

```
Client                           Server
   |                               |
   |-------- SYN (seq=x) -------->|
   |                               |
   |<--- SYN-ACK (seq=y, ack=x+1) -|
   |                               |
   |----- ACK (seq=x+1, ack=y+1) ->|
   |                               |
Connection Established
```

Steps:

1. **Client** → **Server**: SYN segment
    - SYN flag set
    - Initial sequence number (ISN) = x
    - No data
2. **Server** → **Client**: SYN-ACK segment
    - SYN and ACK flags set
    - Server's ISN = y
    - ACK = x + 1 (acknowledging client's SYN)
3. **Client** → **Server**: ACK segment
    - ACK flag set
    - seq = x + 1
    - ACK = y + 1 (acknowledging server's SYN)
    - Can include data

Why 3-way? 2-way insufficient due to delayed/duplicate connection requests

Connection Termination (Four-Way Handshake):

```
Client                          Server
   |                               |
   |-------- FIN (seq=x) -------->|
   |                               |
   |<----- ACK (ack=x+1) ---------|
   |                               |
   |<----- FIN (seq=y) -----------|
   |                               |
   |----- ACK (ack=y+1) --------->|
   |                               |
Connection Closed
```

Steps:

1. Client sends FIN (active close)
2. Server ACKs FIN (half-close: server → client still open)
3. Server sends FIN (when done sending)
4. Client ACKs FIN
5. Client waits 2×MSL (Maximum Segment Lifetime) before closing

TIME_WAIT: Client waits 2×MSL (typically 2-4 minutes) to ensure:

- Server receives final ACK
- Delayed segments from old connection expire

Flow Control (Sliding Window):

- **Receive Window (rwnd)**: Advertised by receiver in Window field
- **Send Window**: Min of congestion window and receive window
- **Zero Window**: Receiver buffers full, sender stops
- **Window Probe**: Sender periodically checks if window opens

Effective Window = rwnd - (LastByteSent - LastByteAcked)

Congestion Control (Network capacity management):

Congestion Window (cwnd): Sender's estimate of network capacity

Sending Rate ≈ cwnd / RTT

Algorithms:

1. Slow Start:

- **Initial**: cwnd = 1 MSS (Maximum Segment Size)
- **Growth**: cwnd doubles every RTT (exponential)
    - For each ACK: cwnd += 1 MSS

- **Continue until**:
  - cwnd ≥ ssthresh (slow start threshold), OR
  - Loss detected

2. Congestion Avoidance:

- **Growth**: cwnd increases by 1 MSS per RTT (linear/additive)
  - For each ACK: cwnd += MSS × (MSS / cwnd)
- **Continue until**: Loss detected

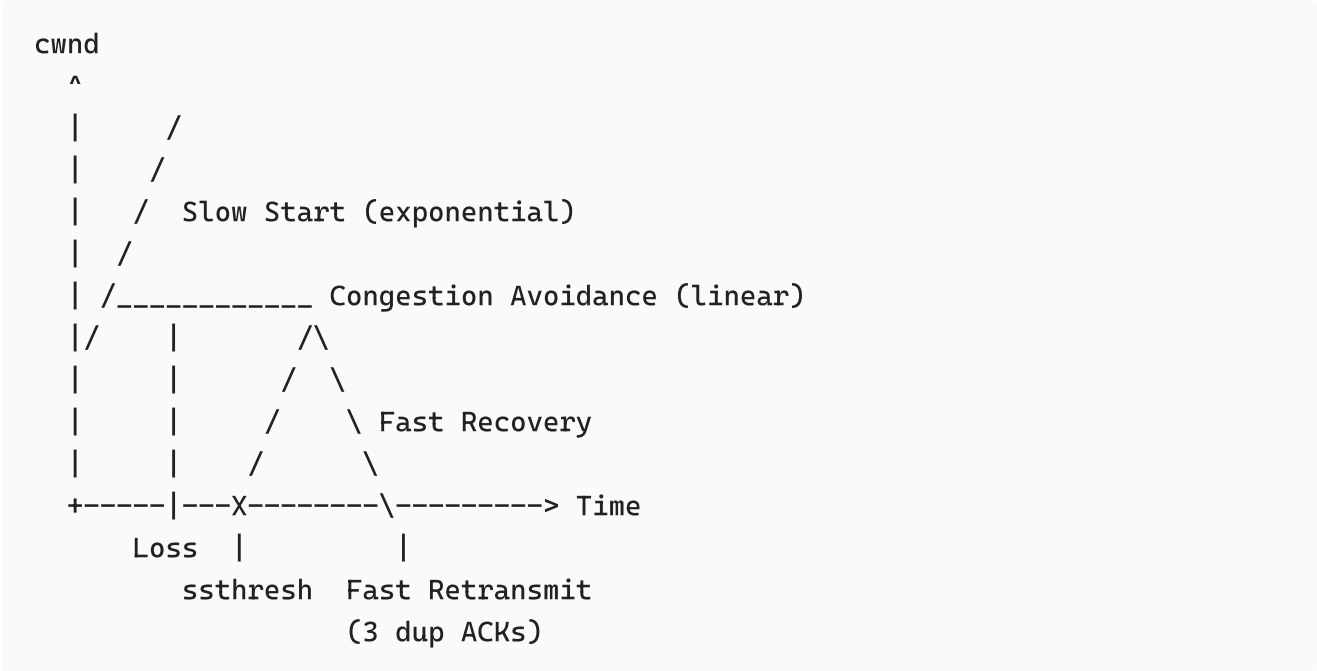
3. Fast Retransmit:

- After **3 duplicate ACKs**: Retransmit without waiting for timeout
- Indicates: Segment lost but network still functioning (not congestion collapse)

4. Fast Recovery (TCP Reno):

- After fast retransmit:
  - ssthresh = cwnd / 2
  - cwnd = ssthresh + 3 MSS
  - Enter congestion avoidance (not slow start)
- Avoids slow start after minor loss

Congestion Control Phases:



On Timeout:

- ssthresh = cwnd / 2
- cwnd = 1 MSS
- Enter slow start

On 3 Duplicate ACKs:

- ssthresh = cwnd / 2
- cwnd = ssthresh + 3
- Fast retransmit + fast recovery

TCP Variants:

1. TCP Tahoe (original):

- Slow start + congestion avoidance
- On loss: cwnd = 1, enter slow start

2. TCP Reno (most common):

- Adds fast retransmit + fast recovery
- Distinguishes between timeout (severe) and 3 dup ACKs (mild)

3. TCP New Reno:

- Improved fast recovery for multiple losses

4. TCP Vegas:

- Proactive congestion avoidance
- Monitors RTT changes

Retransmission Timer:

- **RTO** (Retransmission Timeout): When to retransmit
- **Estimation**: Exponential weighted moving average
  - $RTT_{estimated} = (1 - \alpha) \times RTT_{estimated} + \alpha \times RTT_{sample}$
  - $\alpha = 0.125$ typically
  - $RTO = RTT_{estimated} + 4 \times RTT_{deviation}$
- **Karn's Algorithm**: Don't use RTT of retransmitted segments
- **Exponential Backoff**: Double RTO on each timeout

Cumulative ACKs:

- ACK = n means all bytes up to n-1 received
- ACK n does NOT mean segment n received (means up to byte n-1)

Selective Acknowledgment (SACK):

- Extension to TCP
- Allows receiver to ACK non-contiguous blocks
- More efficient recovery from multiple losses

Silly Window Syndrome:

- Problem: Small segments waste bandwidth (high overhead)
- **Solutions**:
  - **Nagle's Algorithm** (sender): Delay sending until full MSS or all outstanding data ACKed
  - **Clark's Algorithm** (receiver): Don't advertise small window increases

TCP Performance:

- **Throughput** ≈ $\frac{\text{Window Size}}{\text{RTT}}$
- **Bandwidth-Delay Product** (BDP): Capacity of "pipe"
  - BDP = Bandwidth × RTT
  - Optimal window size ≈ BDP
- **Utilization**: $U = \frac{\text{Data Size}}{\text{Data Size}+\text{Overhead}+\text{RTT}}$

Problem-Solving Tips:

- Sequence numbers count bytes, not segments
- ACK number is next expected byte
- SYN and FIN consume one sequence number each
- cwnd in bytes, not segments (usually expressed in MSS units)
- On timeout: Reset to slow start (cwnd = 1)
- On 3 dup ACKs: Fast recovery (cwnd = ssthresh + 3)

GATE Tips:

- 3-way handshake: Client sends SYN, server responds SYN-ACK, client sends ACK
- Connection termination: 4 segments (FIN, ACK, FIN, ACK)
- Slow start: Exponential growth (doubles per RTT)
- Congestion avoidance: Linear growth (additive increase)
- AIMD: Additive Increase Multiplicative Decrease
- On loss: ssthresh = cwnd/2
- Maximum window size with 16-bit field: 65,535 bytes (without window scaling)
- Checksum covers: Pseudo-header + TCP header + data
- Full-duplex: Both directions independent (separate seq/ack)

Example - Sequence Numbers:

```
Initial seq = 1000, MSS = 500 bytes

Segment 1: seq=1000, len=500 (bytes 1000-1499)
ACK: ack=1500
```

```
Segment 2: seq=1500, len=500 (bytes 1500-1999)
ACK: ack=2000

SYN segment: Consumes 1 seq number (even with no data)
```

Example - Congestion Control:

```
Initial: cwnd=1, ssthresh=16

Round 1: cwnd=1 (slow start)
Round 2: cwnd=2
Round 3: cwnd=4
Round 4: cwnd=8
Round 5: cwnd=16 (reached ssthresh)
Round 6: cwnd=17 (congestion avoidance)
Round 7: cwnd=18
...
Timeout at Round 10 (cwnd=20):
  ssthresh = 20/2 = 10
  cwnd = 1 (restart slow start)
```

4.31 Token Bucket (2)

Key Concepts: Rate r, burst b.

4.32 UDP (4)

Key Concepts: Connectionless, no reliability.

5. Databases (284)

5.1 B Tree (31)

Key Concepts: Balanced search tree, order m (2-4 children leaf). **Tips**: Height $log_m n$.

5.2 Candidate Key (5)

Key Concepts: Minimal superkey.

5.3 Conflict Serializable (9)

Key Concepts: Precedence graph acyclic.

5.4 Database Design (1)

Key Concepts: E-R to relational.

5.5 Database Normalization (56)

Key Concepts: Database normalization is the systematic process of organizing data to minimize redundancy and eliminate undesirable characteristics like insertion, update, and deletion anomalies. It decomposes relations into smaller, well-structured relations.

Goals of Normalization:

1. **Eliminate Redundancy**: Reduce duplicate data storage
2. **Prevent Anomalies**: Avoid inconsistencies during updates
3. **Ensure Data Integrity**: Maintain logical consistency
4. **Optimize Storage**: Reduce space requirements
5. **Improve Maintainability**: Simplify database updates

Database Anomalies:

Insertion Anomaly: Cannot insert certain data without having other unrelated data

- **Example**: Cannot add new course without enrolling a student

Update Anomaly: Must update multiple rows for single logical change

- **Example**: Changing instructor name requires updating all course records

Deletion Anomaly: Deleting row causes loss of other valuable information

- **Example**: Dropping last student in course loses course information

Functional Dependencies (FDs):

Definition: $X \rightarrow Y$ (X functionally determines Y) means:

- For any two tuples t_1, t_2: if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$
- Given X value, Y value is uniquely determined
- X is **determinant**, Y is **dependent**

Types of Functional Dependencies:

1. Trivial FD: $X \rightarrow Y$ where $Y \subseteq X$

- Example: $AB \rightarrow A$ (always true)
- **Trivial**: Doesn't provide new information

2. Non-trivial FD: $X \rightarrow Y$ where $Y \not\subset X$

- Example: StudentID → StudentName
- **Meaningful**: Provides semantic constraint

3. Completely Non-trivial FD: $X \rightarrow Y$ where $X \cap Y = \emptyset$

- Example: StudentID → CourseName

- **Strongest form**: No overlap between determinant and dependent

Armstrong's Axioms (Inference Rules):

Primary Axioms (Sound and Complete):

1. **Reflexivity**: If $Y \subseteq X$, then $X \rightarrow Y$
2. **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
3. **Transitivity**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

Derived Rules:
4. **Union**: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
5. **Decomposition**: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
6. **Pseudo-transitivity**: If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$

Proof of Union Rule:

- Given: $X \rightarrow Y$ and $X \rightarrow Z$
- By augmentation on $X \rightarrow Y$: $XZ \rightarrow YZ$
- By augmentation on $X \rightarrow Z$: $XY \rightarrow ZY$
- We have $X \rightarrow Y$, so $X \rightarrow XY$ (by augmentation)
- By transitivity: $X \rightarrow XY$ and $XY \rightarrow ZY$, so $X \rightarrow ZY$
- Since ZY contains both Y and Z: $X \rightarrow YZ$

Closure Operations:

Attribute Closure (X^+):

- Set of all attributes functionally determined by X
- **Algorithm**:

```
Closure(X, F):
    result = X
    while (changes occur):
        for each FD Y → Z in F:
            if Y ⊆ result:
                result = result ∪ Z
    return result
```

Uses of Attribute Closure:

1. **Test if $X \rightarrow Y$**: Check if $Y \subseteq X^+$
2. **Find candidate keys**: X is superkey iff $X^+ = R$
3. **Test equivalence**: $F_1 \equiv F_2$ iff they have same closures

FD Set Closure (F^+):

- Set of all FDs logically implied by F
- **Infinite set** in general
- **Canonical Cover**: Minimal equivalent set

Keys and Superkeys:

Superkey: Set of attributes that functionally determines all attributes

- K is superkey iff $K^+ = R$ (all attributes)
- **Example**: In R(A,B,C) with A → BC, both {A} and {A,B} are superkeys

Candidate Key: Minimal superkey

- No proper subset is a superkey
- **Algorithm to find all candidate keys**:
  1. Start with attributes that appear only on LHS of FDs
  2. Add other attributes one by one until superkey found
  3. Remove redundant attributes

Primary Key: One candidate key chosen as primary identifier

Prime vs Non-Prime Attributes:

- **Prime attribute**: Part of at least one candidate key
- **Non-prime attribute**: Not part of any candidate key
- **Critical for**: 2NF and 3NF definitions

Normal Forms:

First Normal Form (1NF):

Definition: Relation is in 1NF if:

- All attributes contain only atomic (indivisible) values
- No repeating groups or multi-valued attributes
- Each cell contains single value

Violations:

```
Student | Courses
--------|------------------
John    | {Math, Physics}
Mary    | {Chemistry}
```

Conversion to 1NF:

```
Student | Course
--------|----------
John    | Math
John    | Physics
Mary    | Chemistry
```

Second Normal Form (2NF):

Definition: Relation is in 2NF if:

- It is in 1NF, AND
- No non-prime attribute is partially dependent on any candidate key

Partial Dependency: Non-prime attribute depends on proper subset of candidate key

Example Violation:

```
R(StudentID, CourseID, StudentName, CourseName, Grade)
FDs: {StudentID, CourseID} → Grade
     StudentID → StudentName
     CourseID → CourseName
```

Problem: StudentName depends only on StudentID (part of key)

Decomposition:

```
R1(StudentID, StudentName)
R2(CourseID, CourseName)
R3(StudentID, CourseID, Grade)
```

Third Normal Form (3NF):

Definition: Relation is in 3NF if:

- It is in 2NF, AND
- No non-prime attribute is transitively dependent on any candidate key

Alternative Definition: For every FD $X \rightarrow A$ in F^+:

- Either X is a superkey, OR
- A is a prime attribute

Transitive Dependency: $A \rightarrow B$ and $B \rightarrow C$ implies $A \rightarrow C$

Example Violation:

```
R(StudentID, DeptID, DeptName, DeptHead)
FDs: StudentID → DeptID
     DeptID → DeptName
     DeptID → DeptHead
```

Problem: StudentID → DeptID → DeptName (transitive)

Decomposition:

```
R1(StudentID, DeptID)
R2(DeptID, DeptName, DeptHead)
```

Boyce-Codd Normal Form (BCNF):

Definition: Relation is in BCNF if:

- For every FD X → A in F^+ where $A \notin X$:
- X is a superkey

Stricter than 3NF: Eliminates all anomalies

Difference from 3NF: BCNF doesn't allow prime attribute to depend on non-superkey

Example: R(Student, Course, Instructor)

```
FDs: {Student, Course} → Instructor
     Instructor → Course
```

Analysis:

- Candidate keys: {Student, Course}, {Student, Instructor}
- Prime attributes: Student, Course, Instructor
- FD "Instructor → Course" violates BCNF (Instructor not superkey)
- But satisfies 3NF (Course is prime attribute)

BCNF Decomposition:

```
R1(Instructor, Course)
R2(Student, Instructor)
```

Fourth Normal Form (4NF):

Multivalued Dependency (MVD): X →→ Y means:

- For any two tuples with same X value, we can swap their Y values and still have valid tuples

- Y values are independent of other attributes given X

Definition: Relation is in 4NF if:

- It is in BCNF, AND
- For every MVD X →→ Y:
  - Either X →→ Y is trivial (Y ⊆ X or XY = R), OR
  - X is a superkey

Example: R(Student, Course, Hobby)

```
Student | Course  | Hobby
--------|---------|--------
John    | Math    | Chess
John    | Math    | Music
John    | Physics | Chess
John    | Physics | Music
```

MVD: Student →→ Course (courses independent of hobbies)

4NF Decomposition:

```
R1(Student, Course)
R2(Student, Hobby)
```

Fifth Normal Form (5NF/PJNF):

Join Dependency: Relation can be losslessly decomposed into projections

Definition: Relation is in 5NF if:

- It is in 4NF, AND
- Every join dependency is implied by candidate keys

Rarely encountered in practice

Decomposition Theory:

Lossless Join Decomposition:

Definition: Decomposition R → {R_1, R_2, ..., R_n} is lossless if:

- R = R_1 ⋈ R_2 ⋈ ... ⋈ R_n (natural join)
- No spurious tuples generated

Binary Decomposition Test: R → {R_1, R_2} is lossless iff:

- (R_1 ∩ R_2) → R_1, OR

- $(R_1 \cap R_2) \rightarrow R_2$

Dependency Preservation:

Definition: Decomposition preserves dependencies if:

- Every FD in F can be checked within some relation in decomposition
- Union of local FD closures equals original closure

Test: $F^+ = (\cup_i \pi_{ri}(F))^+$

Synthesis Algorithm (3NF with dependency preservation):

1. Find canonical cover of F
2. For each FD $X \rightarrow Y$ in canonical cover, create relation R(XY)
3. If no relation contains candidate key, add one
4. Remove redundant relations

BCNF Decomposition Algorithm:

1. If R not in BCNF, find violating FD $X \rightarrow Y$
2. Decompose into $R_1(XY)$ and $R_2(R - Y + X)$
3. Recursively apply to R_1 and R_2
4. Result: Lossless join, may not preserve dependencies

Canonical Cover (Minimal Set of FDs):

Definition: Canonical cover Fc of F is minimal set such that $F^+ = Fc^+$

Properties:

1. **Right-hand sides are single attributes**: $X \rightarrow A$ (not $X \rightarrow AB$)
2. **No extraneous attributes**: Cannot remove attribute from LHS
3. **No redundant FDs**: Cannot remove entire FD

Algorithm:

```
CanonicalCover(F):
1. Decompose RHS: Replace X → YZ with X → Y, X → Z
2. Remove extraneous attributes from LHS:
   For each FD X → A and each attribute B in X:
       If (X - {B})⁺ contains A, remove B from X
3. Remove redundant FDs:
   For each FD X → A:
       If A ∈ (F - {X → A})⁺, remove X → A
4. Return resulting set
```

Example:

```
F = {A → BC, B → C, A → B, AB → C}

Step 1: {A → B, A → C, B → C, A → B, AB → C}
Step 2: AB → C becomes A → C (B extraneous)
Step 3: Remove A → C (redundant, follows from A → B, B → C)
Step 4: Remove duplicate A → B

Canonical Cover: {A → B, B → C}
```

Advanced Concepts:

Multivalued Dependencies (MVDs):

Definition: $X \twoheadrightarrow Y$ (X multi-determines Y) if:

- In any relation instance, for tuples t_1, t_2 with $t_1[X] = t_2[X]$,
- There exist tuples t_3, t_4 such that:
  - $t_3[X] = t_4[X] = t_1[X] = t_2[X]$
  - $t_3[Y] = t_1[Y]$, $t_3[R\text{-}X\text{-}Y] = t_2[R\text{-}X\text{-}Y]$
  - $t_4[Y] = t_2[Y]$, $t_4[R\text{-}X\text{-}Y] = t_1[R\text{-}X\text{-}Y]$

Intuition: Y values are independent of (R-X-Y) values given X

Properties of MVDs:

1. **Complementation**: $X \twoheadrightarrow Y$ iff $X \twoheadrightarrow (R\text{-}X\text{-}Y)$
2. **FD implies MVD**: If $X \rightarrow Y$, then $X \twoheadrightarrow Y$
3. **Trivial MVD**: $X \twoheadrightarrow Y$ is trivial if $Y \subseteq X$ or $XY = R$

Join Dependencies:

Definition: Relation R satisfies join dependency $\bowtie\{R_1, R_2, ..., R_n\}$ if:

- $R = \pi_{r1}(R) \bowtie \pi_{r2}(R) \bowtie ... \bowtie \pi_{rn}(R)$

Trivial Join Dependency: One of $R_i = R$

Normalization Algorithms:

3NF Synthesis Algorithm:

```
3NF_Synthesis(R, F):
1. Compute canonical cover Fc of F
2. For each FD X → A in Fc:
     Create relation schema (X ∪ {A})
3. If no schema contains a candidate key of R:
     Create additional schema containing any candidate key
```

```
4. Remove redundant schemas (if one is subset of another)
5. Return collection of schemas
```

Properties:

- **Lossless join**: Yes
- **Dependency preservation**: Yes
- **Normal form**: 3NF guaranteed

BCNF Decomposition Algorithm:

```
BCNF_Decomposition(R, F):
1. If R is in BCNF, return {R}
2. Find FD X → Y that violates BCNF
3. Decompose R into:
     R1 = X ∪ Y
     R2 = R - Y + X
4. Recursively apply to R1 and R2
5. Return union of results
```

Properties:

- **Lossless join**: Yes
- **Dependency preservation**: Not guaranteed
- **Normal form**: BCNF guaranteed

Testing Decomposition Properties:

Lossless Join Test:

```
LosslessJoinTest(R, {R1, R2}, F):
1. Create table with |F| rows and |R| columns
2. For each FD X → Y:
     If Ri contains X ∪ Y, put aⱼ in row i, column j
     Otherwise, put bᵢⱼ
3. Apply FD rules until no changes
4. Lossless iff some row becomes all a's
```

Dependency Preservation Test:

```
DependencyPreservationTest(R, {R1, R2, ..., Rn}, F):
1. For each FD X → Y in F:
     Compute X⁺ using only FDs that can be checked in some Ri
2. If Y ⊆ X⁺ for all FDs, then dependency preserving
```

Practical Normalization Process:

Step 1: Identify Entities and Attributes

- Determine what real-world objects are represented
- List all attributes and their domains
- Identify natural groupings

Step 2: Determine Functional Dependencies

- Analyze business rules and constraints
- Identify determinant-dependent relationships
- Consider semantic meaning of attributes

Step 3: Find Candidate Keys

- Use attribute closure to identify superkeys
- Find minimal superkeys (candidate keys)
- Choose primary key

Step 4: Check Normal Forms

- Start with 1NF (atomic values)
- Progress through 2NF, 3NF, BCNF as needed
- Apply appropriate decomposition algorithms

Step 5: Verify Decomposition

- Test for lossless join property
- Check dependency preservation
- Ensure no information loss

Trade-offs in Normalization:

Benefits of Higher Normal Forms:

- Reduced redundancy
- Eliminated anomalies
- Better data integrity
- Easier maintenance

Costs of Higher Normal Forms:

- More relations (tables)
- More complex queries (joins)
- Potential performance overhead
- Increased storage for foreign keys

Denormalization:

When to Denormalize:

- Read-heavy workloads
- Performance critical applications
- Data warehouse scenarios
- When joins are expensive

Controlled Denormalization:

- Maintain normalized base tables
- Create denormalized views or materialized views
- Use triggers to maintain consistency

Problem-Solving Strategy:

Normalization Problems:

1. **Identify FDs**: From problem description or given
2. **Find candidate keys**: Use attribute closure
3. **Classify attributes**: Prime vs non-prime
4. **Check normal forms**: Apply definitions systematically
5. **Decompose if needed**: Use appropriate algorithm
6. **Verify properties**: Lossless join, dependency preservation

Key Finding:

1. **Start with LHS-only attributes**: Must be in every key
2. **Add others systematically**: Until closure covers all attributes
3. **Minimize**: Remove redundant attributes

GATE Tips:

- **1NF**: Atomic values only (no sets, arrays, nested tables)
- **2NF**: No partial dependencies (only matters for composite keys)
- **3NF**: No transitive dependencies (non-prime → non-prime)
- **BCNF**: Every determinant is superkey (stricter than 3NF)
- **4NF**: No non-trivial MVDs
- **Candidate key**: Minimal superkey
- **Prime attribute**: Part of some candidate key
- **Lossless decomposition**: Common attributes form superkey
- **Dependency preservation**: All FDs checkable locally
- **3NF synthesis**: Always preserves dependencies
- **BCNF decomposition**: Always lossless, may lose dependencies

Common GATE Examples:

Example 1: Find candidate keys

```
R(A, B, C, D, E)
F = {A → BC, CD → E, B → D, E → A}

Step 1: No LHS-only attributes
Step 2: Try single attributes:
  A⁺ = {A, B, C, D, E} = R ✓ (A is candidate key)
  E⁺ = {E, A, B, C, D} = R ✓ (E is candidate key)

Candidate keys: {A}, {E}
```

Example 2: Check normal forms

```
R(A, B, C, D)
F = {AB → C, C → D, D → A}
Candidate keys: {AB}, {BC}

2NF: No partial dependencies ✓
3NF: C → D violates (C not superkey, D non-prime) ✗
BCNF: Multiple violations ✗

Decomposition for 3NF:
R1(C, D), R2(A, B, C)
```

Example 3: Lossless join test

```
R(A, B, C), F = {A → B}
Decomposition: R1(A, B), R2(A, C)

Test: R1 ∩ R2 = {A}
Need: A → R1 or A → R2
A⁺ = {A, B} = R1 ✓

Lossless join decomposition
```

Example 4: Canonical cover

```
F = {A → BC, B → C, A → B, AB → C}

Step 1: Decompose RHS
{A → B, A → C, B → C, A → B, AB → C}

Step 2: Remove extraneous attributes
AB → C: B is extraneous (A⁺ contains C)
Result: {A → B, A → C, B → C, A → B, A → C}
```

```
Step 3: Remove redundant FDs
A → C is redundant (A → B, B → C gives A → C)
A → B appears twice

Canonical Cover: {A → B, B → C}
```

5.6 Decomposition (1)

Key Concepts: Lossless if R1 ∩ R2 → keys.

5.7 ER Diagram (12)

Key Concepts: Entities, relationships cardinalities.

5.8 Functional Dependency (2)

Key Concepts: X→Y if determines.

5.9 Indexing (13)

Key Concepts: B+ tree for range.

5.10 Joins (7)

Key Concepts: Natural, theta.

5.11 Multivalued Dependency 4nf (1)

Key Concepts: MVD X→→Y if independent.

5.12 Natural Join (3)

Key Concepts: On common attrs.

5.13 Query (1)

Key Concepts: SQL select.

5.14 Referential Integrity (5)

Key Concepts: FK references PK.

5.15 Relational Algebra (31)

Key Concepts: Select $sigma$, project pi, join $bowtie$.

5.16 Relational Calculus (15)

Key Concepts: Tuple $tmidP(t)$, domain.

5.17 Relational Model (2)

Key Concepts: Tables, keys.

5.18 SQL (58)

Key Concepts: Group by having; subqueries. **Tips**: Correlated vs non.

5.19 Transaction and Concurrency (30)

Key Concepts: Ensures database consistency when multiple transactions execute concurrently. Balances consistency with performance.

ACID Properties:

1. **Atomicity**: All or nothing (transaction is indivisible unit)
   - If failure occurs, rollback to initial state
   - Implemented via logging and recovery
2. **Consistency**: Database moves from one valid state to another
   - Integrity constraints maintained
   - Application responsibility to ensure valid transactions
3. **Isolation**: Concurrent transactions don't interfere
   - Appears as if transactions execute serially
   - Implemented via concurrency control
4. **Durability**: Committed changes persist (survive crashes)
   - Write to stable storage (disk)
   - Implemented via logging and recovery

Transaction Operations:

- **Read(X)**: Read data item X
- **Write(X)**: Write data item X
- **Commit**: Transaction completes successfully
- **Abort**: Transaction fails, rollback changes

Schedules:

- **Serial Schedule**: Transactions execute one after another
  - Always correct but low concurrency
  - For n transactions: n! possible serial schedules
- **Concurrent Schedule**: Operations from different transactions interleaved
  - Higher throughput but risk of inconsistency

Serializability:

A concurrent schedule is **serializable** if equivalent to some serial schedule.

1. Conflict Serializability (most common in GATE):

Conflicting Operations: Two operations conflict if:

- From different transactions
- Access same data item
- At least one is Write

Conflict types:

- **Read-Write (RW)**: $R_i(X), W_j(X)$
- **Write-Read (WR)**: $W_i(X), R_j(X)$
- **Write-Write (WW)**: $W_i(X), W_j(X)$

Non-conflicting: $R_i(X), R_j(X)$ (both reads)

Testing Conflict Serializability (Precedence Graph):

1. Create node for each transaction
2. Add edge $T_i \rightarrow T_j$ if T_i has operation that conflicts with T_j's operation and comes before it
3. Schedule is conflict serializable $\Leftrightarrow$ Graph is **acyclic** (DAG)
4. Topological sort gives equivalent serial order

Example:

```
T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)

Schedule: R1(A), R2(A), W1(A), R1(B), W2(A), R2(B), W1(B), W2(B)

Conflicts:
- R1(A) < W2(A): T1 → T2
- W1(A) < W2(A): T1 → T2
- W1(B) < W2(B): T1 → T2

Graph: T1 → T2 (acyclic) ✓ Conflict serializable
Equivalent to: T1, T2
```

2. View Serializability:

Two schedules are view equivalent if:

- Same initial reads (if T_i reads X initially in S1, same in S2)
- Same updates are read (if T_i reads X written by T_j in S1, same in S2)
- Same final writes (if T_i writes X last in S1, same in S2)
- View serializable $\supseteq$ Conflict serializable

- Testing view serializability is NP-complete
- Rarely asked in GATE

Concurrency Control Protocols:

1. Lock-Based Protocols:

Basic Locking:

- **Shared Lock (S)**: For Read, multiple transactions can hold
- **Exclusive Lock (X)**: For Write, only one transaction can hold

Compatibility matrix:

```
     S   X
S    ✓   ✗
X    ✗   ✗
```

Two-Phase Locking (2PL) - Ensures conflict serializability:

- **Growing Phase**: Acquire locks, cannot release
- **Shrinking Phase**: Release locks, cannot acquire
- **Lock point**: When last lock acquired

Issues with Basic 2PL:

- **Cascading Rollback**: If T1 aborts, all transactions that read T1's writes must abort
- **Deadlock**: Possible (need detection/prevention)

Strict 2PL (most common):

- Hold all exclusive locks until commit/abort
- **Prevents**: Cascading rollback, dirty reads
- **Ensures**: Conflict serializability + recoverability

Rigorous 2PL:

- Hold ALL locks (shared + exclusive) until commit/abort
- Simplest to implement

2. Timestamp-Based Protocols:

- Each transaction T_i has unique timestamp $TS(T_i)$
- For each data item X, maintain:
  - $R\_TS(X)$: Largest timestamp of any read
  - $W\_TS(X)$: Largest timestamp of any write

Basic Timestamp Ordering:

- **Read(X)**:
  - If $TS(T_i) < W\_TS(X)$: Abort T_i (reading obsolete)
  - Else: Execute, set $R\_TS(X) = \max(R\_TS(X), TS(T_i))$
- **Write(X)**:
  - If $TS(T_i) < R\_TS(X)$: Abort T_i (writing obsolete)
  - If $TS(T_i) < W\_TS(X)$: Abort T_i (overwriting newer)
  - Else: Execute, set $W\_TS(X) = TS(T_i)$
- **Ensures**: Conflict serializability
- **No deadlocks** (timestamp order determines precedence)
- **Starvation possible** (repeated aborts)

3. Optimistic Concurrency Control (Validation-Based):

Phases:

1. **Read**: Transaction reads and computes in private workspace
2. **Validate**: Check if conflicts with other transactions
3. **Write**: If valid, apply changes to database

- Good for read-mostly workloads
- Low overhead if conflicts rare

4. Multi-Version Concurrency Control (MVCC):

- Keep multiple versions of each data item
- Reads never blocked by writes
- Used in PostgreSQL, Oracle

Isolation Levels (SQL standard):

1. **Read Uncommitted**: Lowest isolation
   - Can read uncommitted changes (dirty reads)
2. **Read Committed**:
   - No dirty reads
   - Non-repeatable reads possible
3. **Repeatable Read**:
   - Same read twice returns same value
   - Phantom reads possible (new rows inserted)
4. **Serializable**: Highest isolation
   - Equivalent to serial execution
   - No anomalies

Deadlock:

Cycle in wait-for graph: T_1 waits for T_2, T_2 waits for T_1

Deadlock Handling:

1. **Prevention**:
   - Wait-Die: Older waits, younger dies
   - Wound-Wait: Older wounds younger, younger waits
2. **Detection**:
   - Maintain wait-for graph
   - Periodically check for cycles
   - If found, abort victim transaction
3. **Timeout**: Abort if waiting too long

Recoverability:

Recoverable Schedule: If T_j reads from T_i, then T_i commits before T_j commits

- Prevents reading uncommitted data that might be rolled back

Cascadeless Schedule (Avoids Cascading Rollback):

- Transaction reads only committed data
- Stronger than recoverable
- Strict 2PL ensures cascadeless

Strict Schedule:

- Don't read/write X until last transaction that wrote X commits/aborts
- Strongest form

Hierarchy: Strict ⊂ Cascadeless ⊂ Recoverable ⊂ All Schedules

Problem-Solving Strategy:

1. **Serializability**: Draw precedence graph, check for cycles
2. **2PL**: Identify lock point, check if shrinking phase follows growing
3. **Deadlock**: Draw wait-for graph, look for cycles
4. **Recoverability**: Check commit order vs read order

GATE Tips:

- Conflict serializable → View serializable (not vice versa)
- 2PL ensures conflict serializability but **doesn't prevent deadlock**
- Strict 2PL ensures cascadeless + conflict serializable
- Precedence graph cycle → Not conflict serializable

- All serial schedules are serializable
- For n transactions: n! serial schedules
- Timestamp ordering: No deadlock but possible starvation

Common Anomalies (if isolation violated):

1. **Dirty Read**: Read uncommitted data
2. **Non-repeatable Read**: Different values on repeated reads
3. **Phantom Read**: New rows appear in repeated range query
4. **Lost Update**: Concurrent writes, one overwrites other

5.20 Tuple Relational Calculus (1)

(See 5.16)

6. Digital Logic (303)

6.1 Adder (9)

Key Concepts: Half: $S = aoplusb$, $C = ab$; full $+c_{textin}$.

6.2 Array Multiplier (2)

Key Concepts: Braun/Booth for n-bit.

6.3 Boolean Algebra (34)

Key Concepts: Idempotent $a + a = a$. **Tips**: Consensus theorem.

6.4 Booths Algorithm (7)

Key Concepts: Recode for mult; shift-add.

6.5 Canonical Normal Form (9)

Key Concepts: SOP/POS from truth table.

6.6 Carry Generator (2)

Key Concepts: In CLA: $G = ab$.

6.7 Circuit Output (40)

Key Concepts: Simplify K-map.

6.8 Combinational Circuit (2)

Key Concepts: No memory.

6.9 Decoder (3)

Key Concepts: 2^n lines from n bits.

6.10 Digital Circuits (6)

Key Concepts: TTL/CMOS levels.

6.11 Digital Counter (17)

Key Concepts: Ripple vs synchronous.

6.12 Finite State Machines (4)

Key Concepts: Moore/Mealy output.

6.13 Fixed Point Representation (2)

Key Concepts: Qm.n format.

6.14 Flip Flop (6)

Key Concepts: SR, JK, D, T; race hazards.

6.15 Floating Point Representation (12)

Key Concepts: IEEE 754: sign, exp bias 127, mantissa 1.f.

6.16 Functional Completeness (7)

Key Concepts: {NAND} or {NOR} universal.

6.17 IEEE Representation (8)

(See 6.15)

6.18 K Map (20)

Key Concepts: Group 1s powers of 2; don’t cares.

6.19 Memory Interfacing (5)

(See 2.15)

6.20 Min No Gates (4)

Key Concepts: NAND/NOR universal min.

6.21 Min Products of Sum Form (2)

Key Concepts: Maxterms.

6.22 Min Sum of Products Form (15)

Key Concepts: Minterms.

6.23 Multiplexer (14)

Key Concepts: 2^n select lines.

6.24 Number Representation (58)

Key Concepts: 2s complement $-x = simx + 1$.

6.25 Prime Implicants (2)

Key Concepts: Essential for Quine-McCluskey.

6.26 ROM (4)

Key Concepts: Lookup table.

6.27 Ripple Counter Operation (1)

Key Concepts: Async, delay prop.

6.28 Sequential Circuit (1)

Key Concepts: With feedback.

6.29 Shift Registers (2)

Key Concepts: SISO, PIPO.

6.30 Static Hazard (1)

Key Concepts: Glitch in combinational.

6.31 Synchronous Asynchronous Circuits (4)

Key Concepts: Clocked vs not.

7. Operating System (335)

7.1 Context Switch (4)

Key Concepts: Save/restore registers; time overhead.

7.2 Deadlock Prevention Avoidance Detection (4)

Key Concepts: Banker’s safe seq; detection via resource alloc graph.

7.3 Disk (31)

Key Concepts: Seek + rot + transfer time.

7.4 Disk Scheduling (14)

Key Concepts: FCFS, SSTF, SCAN, C-SCAN.

7.5 File System (9)

Key Concepts: FAT, inode.

7.6 Fork System Call (7)

Key Concepts: Child copy of parent.

7.7 IO Handling (7)

(See 2.9)

7.8 Input Output (1)

(See IO)

7.9 Inter Process Communication (1)

Key Concepts: Shared mem, pipes, msg queues.

7.10 Interrupts (6)

(See 2.13)

7.11 Linked Allocation (1)

Key Concepts: Pointers in blocks; external frag.

7.12 Memory Management (10)

Key Concepts: Paging, segmentation.

7.13 Multilevel Paging (1)

Key Concepts: Page table of page tables.

7.14 OS Protection (3)

Key Concepts: Access control lists.

7.15 Optimal Page Replacement (1)

Key Concepts: Belady: farthest future ref.

7.16 Page Replacement (33)

Key Concepts: FIFO, LRU, OPT; Belady anomaly.

7.17 Precedence Graph (3)

(See 5.3)

7.18 Process (5)

Key Concepts: PCB, states ready/run/wait.

7.19 Process Scheduling (48)

Key Concepts: Determines which process runs on CPU. Goals: CPU utilization, throughput, minimize turnaround/waiting/response time, fairness.

Performance Metrics:

- **Arrival Time (AT)**: When process enters ready queue
- **Burst Time (BT)**: CPU time required
- **Completion Time (CT)**: When process finishes
- **Turnaround Time (TAT)**: CT - AT (total time in system)
- **Waiting Time (WT)**: TAT - BT (time in ready queue)
- **Response Time (RT)**: First CPU time - AT (for interactive systems)
- **Throughput**: Processes completed per unit time
- **CPU Utilization**: % time CPU is busy

Scheduling Algorithms:

1. First Come First Serve (FCFS):

- Non-preemptive, simple FIFO queue
- Pros: Simple, no starvation
- Cons: Convoy effect (short jobs wait for long job)
- Average WT can be high
- Example: P1(24), P2(3), P3(3) → Avg WT = (0+24+27)/3 = 17

2. Shortest Job First (SJF):

- Non-preemptive: Choose shortest burst time
- **Optimal** for average waiting time (provable)
- Cons: Starvation possible, need to predict burst time
- Prediction: Exponential averaging $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
- Example: P1(6), P2(8), P3(7), P4(3) → Order: P4, P1, P3, P2

3. Shortest Remaining Time First (SRTF/SRTN):

- Preemptive version of SJF

- Preempt if new process has shorter remaining time
- Optimal for average waiting time (preemptive category)
- Cons: More context switches, starvation
- Example: New short job can preempt currently running job

4. Round Robin (RR):

- Preemptive, time quantum/slice (typically 10-100ms)
- Each process gets equal CPU time in circular queue
- After quantum expires, move to back of queue
- Pros: Fair, good response time, no starvation
- Cons: Higher turnaround than SJF
- Performance depends on quantum:
  - Too large: Degenerates to FCFS
  - Too small: Too many context switches (overhead)
  - Rule of thumb: 80% bursts should be shorter than quantum
- Context switch time should be << quantum

5. Priority Scheduling:

- Each process has priority (lower number = higher priority)
- Can be preemptive or non-preemptive
- Pros: Important processes get preference
- Cons: **Starvation** of low-priority processes
- Solution: **Aging** - increase priority over time
- Priority inversion: Low priority holds resource needed by high priority

6. Multilevel Queue:

- Multiple ready queues with different priorities
- Each queue has own scheduling algorithm
- Example: Foreground (RR), Background (FCFS)
- Fixed priority between queues or time slicing
- No movement between queues

7. Multilevel Feedback Queue:

- Like multilevel queue but processes can move between queues
- Typically: Higher priority queues have smaller quantum
- I/O-bound and interactive processes stay in higher queues
- CPU-bound processes demoted to lower queues
- Pros: Flexible, favors I/O bound
- Most general, configurable

8. Highest Response Ratio Next (HRRN):

- Non-preemptive
- Priority = $\frac{\text{Waiting time}+\text{Burst time}}{\text{Burst time}} = \frac{TAT}{BT}$
- Favors shorter jobs but ages longer jobs
- No starvation

Preemptive vs Non-Preemptive:

- **Preemptive**: OS can forcibly remove process from CPU
  - Better for time-sharing, interactive systems
  - More context switches, overhead
  - Examples: RR, SRTF, Priority (preemptive)
- **Non-preemptive**: Process runs until completion/blocks
  - Simpler, less overhead
  - Risk of process monopolizing CPU
  - Examples: FCFS, SJF, Priority (non-preemptive)

Problem-Solving Steps:

1. Draw Gantt chart showing process execution timeline
2. Calculate CT for each process from chart
3. TAT = CT - AT for each process
4. WT = TAT - BT for each process
5. Average = sum/n

GATE Tips:

- SJF gives minimum average WT (non-preemptive)
- SRTF gives minimum average WT (preemptive)
- FCFS can have convoy effect
- RR with quantum → ∞ becomes FCFS
- Preemptive scheduling needed for real-time systems
- For RR, if all processes arrive at t=0 with same BT, order doesn't matter

Example Problem:

Processes: P1(AT=0,BT=8), P2(AT=1,BT=4), P3(AT=2,BT=9), P4(AT=3,BT=5)

FCFS: P1→P2→P3→P4

- TAT: 8, 11, 18, 23 → Avg = 15
- WT: 0, 7, 9, 15 → Avg = 7.75

SRTF:

- t=0: P1(8)
- t=1: P2(4) preempts P1(7 remaining)
- t=5: P1(7) resumes
- t=8: P4(5) preempts P1(4 remaining)
- ...
- Avg WT: Much lower than FCFS

RR (quantum=4):

- Queue rotation: P1(4)→P2(4)→P1(4)→P3(4)→P4(4)→P3(4)→P4(1)→P3(1)
- More context switches, fair distribution

7.20 Process Synchronization (52)

Key Concepts: Peterson, bakery; critical section.

7.21 Resource Allocation (27)

Key Concepts: Matrix for max/alloc/avail.

7.22 Resource Allocation Graph (1)

Key Concepts: Cycles for deadlock.

7.23 Semaphore (10)

Key Concepts: Binary/mutex, counting; wait/signal.

7.24 Srtf (1)

Key Concepts: Preemptive SJF.

7.25 System Calls (1)

Key Concepts: Read/write/fork.

7.26 Threads (10)

Key Concepts: User/kernel, lightweight processes.

7.27 Virtual Memory (43)

(See 2.20; thrashing when working set < frame}.

8. Programming and DS: Data Structures (236)

8.1 AVL Tree (6)

Key Concepts: Height-balanced BST, rotations. **Tips**: BF $=$ height(left) $-$ height(right) $in-1,0,1.$

8.2 Array (13)

Key Concepts: Fixed size, $O(1)$ access.

8.3 Binary Heap (29)

Key Concepts: Min/max, complete tree; insert $O(logn)$.

8.4 Binary Search Tree (36)

Key Concepts: Inorder sorted; search $O(h)$.

8.5 Binary Tree (53)

Key Concepts: Full/proper; traversal preorder/in/post.

8.6 Data Structures (5)

General

8.7 Hashing (16)

(See 1.16)

8.8 Infix Prefix (5)

Key Concepts: Shunting-yard for conversion.

8.9 Linked List (23)

Key Concepts: Singly/doubly; insert $O(1)$ with ptr.

8.10 Number of Swap (1)

Key Concepts: In bubble sort $O(n^2)$.

8.11 Priority Queue (2)

Key Concepts: Heap impl.

8.12 Queue (14)

Key Concepts: FIFO, circular array.

8.13 Stack (18)

Key Concepts: LIFO, recursion.

8.14 Time Complexity (1)

(See 1.34)

8.15 Tree (14)

(See binary)

9. Programming: Programming in C (127)

9.1 Aliasing (1)

Key Concepts: Multiple ptrs same mem.

9.2 Array (10)

(See 8.2)

9.3 Functions (1)

Key Concepts: Declaration, definition.

9.4 Goto (2)

Key Concepts: Labels, spaghetti code.

9.5 Identify Function (6)

(See 1.18)

9.6 Loop Invariants (8)

Key Concepts: True before/after iteration.

9.7 Output (10)

Key Concepts: printf formats %d %s.

9.8 Parameter Passing (12)

(See 3.16)

9.9 Pointers (14)

Key Concepts: *p, &x; arithmetic p+i = p + i*sizeof.

9.10 Programming Constructs (1)

If/while/for

9.11 Programming In C (31)

General syntax

9.12 Programming Paradigms (2)

- *Procedural vs OO.

9.13 Recursion (18)

(See 1.28; tail opt)

9.14 Strings (1)

Null-term char arrays

9.15 Structure (5)

Key Concepts: struct tag {types}; sizeof.

9.16 Switch Case (2)

Key Concepts: Fall-through.

9.17 Type Checking (1)

- *Static/dynamic.

9.18 Union (1)

Key Concepts: Overlapping mem.

9.19 Variable Binding (1)

(See 3.23)

10. Theory of Computation (286)

10.1 Closure Property (9)

Key Concepts: Regular langs closed under $\cup$, *, concat.

10.2 Context Free Grammar (2)

Key Concepts: $S \to \alpha$, $|\alpha| \geq 1$ nonterm.

10.3 Context Free Language (33)

Key Concepts: Pumpable; PDA accept.

10.4 Countable Uncountable Set (3)

(See 4.2)

10.5 Decidability (29)

Key Concepts: Determines whether problems are algorithmically solvable. Fundamental limits of computation.

Definitions:

Decidable (Recursive) Language:

- A language L is decidable if ∃ Turing Machine (TM) that:
  - **Halts** on all inputs
  - **Accepts** if $w \in L$
  - **Rejects** if $w \notin L$
- Also called: Recursive, Computable, Solvable

Recognizable (Recursively Enumerable) Language:

- A language L is recognizable if ∃ TM that:
  - **Accepts** if $w \in L$
  - **Rejects or loops forever** if $w \notin L$
- May not halt on inputs not in language
- Also called: RE, Semi-decidable, Turing-recognizable

Relationships:

- Decidable ⊂ RE (every decidable language is RE)
- Co-RE: Complement of RE language
- **Decidable** ⟺ Both L and $\overline{L}$ are RE
- If L is RE but not decidable → $\overline{L}$ is not RE

Hierarchy:

```
All Languages
  |
  |-- Recursively Enumerable (RE)
  |     |
  |     |-- Decidable (Recursive)
  |     |     |
  |     |     |-- Context-Sensitive
  |     |     |     |
  |     |     |     |-- Context-Free
  |     |     |     |     |
  |     |     |     |     |-- Regular
```

Undecidable Problems (No algorithm exists):

1. Halting Problem (Most famous):

- **Problem**: Given TM M and input w, does M halt on w?
- **Proof** (Diagonalization by Turing):

```
Assume HALT(M, w) decides halting problem
Construct Paradox(M):
  If HALT(M, M) accepts:
    Loop forever
  Else:
    Halt

What does Paradox(Paradox) do?
- If it halts → HALT says it halts → Loop (contradiction!)
- If it loops → HALT says it loops → Halt (contradiction!)

Therefore, HALT cannot exist.
```

- **Consequence**: Most fundamental undecidable problem

2. Rice's Theorem (Powerful generalization):

- **Statement**: Any non-trivial property of RE languages is undecidable
- **Non-trivial property**:
  - True for some RE languages
  - False for some RE languages
  - Concerns language, not encoding
- **Examples** (all undecidable):
  - Does L(M) = ∅? (Emptiness)
  - Is L(M) finite? (Finiteness)
  - Is L(M) regular?
  - Does L(M) = Σ*? (Totality)
  - Does $L(M_1) = L(M_2)$? (Equivalence)
  - Does $L(M_1) \subseteq L(M_2)$? (Containment)

3. Post Correspondence Problem (PCP):

- Given: Tiles with top and bottom strings
- Question: Can we arrange tiles to make top = bottom?
- Example:

```
Tiles: [a/ab], [b/ca], [ca/a], [abc/c]
Solution: [ca/a][a/ab][abc/c] → ca·a·abc = a·ab·c? NO
```

- Undecidable for general case
- Decidable for 2 tiles

4. Ambiguity Problem:

- Given CFG G, is G ambiguous?
- Undecidable

5. Totality Problem:

- Does TM M accept all inputs?
- Undecidable

Decidable Problems (Algorithm exists):

For Regular Languages (all decidable):

- **Membership**: Is $w \in L$? (Simulate DFA/NFA)
- **Emptiness**: Is L = ∅? (Check if accept state reachable)
- **Finiteness**: Is L finite? (Check for cycles to accept states)
- **Equivalence**: Is $L(M_1) = L(M_2)$? (Minimize DFAs, compare)
- **Containment**: Is $L(M_1) \subseteq L(M_2)$?
- **Intersection**: Is $L(M_1) \cap L(M_2) = \emptyset$?

For Context-Free Languages:

- **Membership**: Is $w \in L$? ✓ (CYK algorithm $O(n^3)$)
- **Emptiness**: Is L = ∅? ✓ (Check if start symbol generating)
- **Finiteness**: Is L finite? ✓ (Check for cycles in parse tree)
- **Equivalence**: Is $L(G_1) = L(G_2)$? ✗ Undecidable
- **Containment**: Is $L(G_1) \subseteq L(G_2)$? ✗ Undecidable
- **Intersection emptiness**: Is $L(G_1) \cap L(G_2) = \emptyset$? ✗ Undecidable
- **Ambiguity**: Is G ambiguous? ✗ Undecidable
- **Regularity**: Is L(G) regular? ✗ Undecidable
- **CFG intersection with Regular**: Decidable (result is CFL)

Reductions (Proving Undecidability):

Reduction: A ≤ B (A reduces to B)

- If we can solve B → We can solve A
- If A undecidable → B undecidable

Method:

1. Choose known undecidable problem A (often Halting)
2. Assume B is decidable
3. Show how to solve A using B (construct reduction)
4. Contradiction → B is undecidable

Example - Emptiness is Undecidable for TMs:

```
Halting Problem: Does M halt on w?
Assume EMPTY decides: Is L(M') = ∅?

Reduction:
Construct M':
  On input x:
    Simulate M on w
    If M halts: Accept x

L(M') = ∅ ⇔ M doesn't halt on w
L(M') = Σ* ⇔ M halts on w

EMPTY(M') would solve Halting Problem!
Contradiction → EMPTY doesn't exist.
```

Turing Machine Acceptance:

Acceptance Types:

1. **Acceptance by Halting**: TM halts in accept state
2. **Acceptance by Final State**: TM ends in designated accept state
3. **Recognition**: TM accepts inputs in language (may loop on others)

TM vs DFA/PDA:

- **DFA**: Always halts (finite states, finite input)
- **PDA**: Always halts (can't loop infinitely on finite input)
- **TM**: May loop forever (unbounded computation)

Church-Turing Thesis:

- Informal hypothesis: "Any effectively computable function can be computed by a Turing Machine"
- Not a theorem (can't be proved)
- Widely accepted
- Equivalent models: Lambda calculus, recursive functions, etc.

Closure Properties:

Decidable Languages (closed under):

- Union, Intersection, Complement
- Concatenation, Kleene star
- All regular operations

RE Languages (closed under):

- Union: Simulate both TMs in parallel
- Intersection: Simulate both sequentially
- Concatenation, Kleene star

RE Languages NOT closed under:

- **Complement**: If RE closed under complement → All RE would be decidable (contradiction)

Problem Classification:

Decidable:

- Regular language problems
- CFL: Membership, emptiness, finiteness
- Arithmetic on natural numbers
- Propositional logic satisfiability (SAT is decidable but NP-complete)

Undecidable but RE:

- Halting problem
- Emptiness for TMs
- Equivalence for TMs
- Properties covered by Rice's theorem

Not even RE:

- Complement of Halting problem
- Non-halting problem: Does M NOT halt on w?
- Totality problem complement

Gödel's Incompleteness Theorem (Related):

- Any sufficiently powerful formal system is either:
  - Incomplete (some true statements unprovable), OR
  - Inconsistent (can prove contradictions)
- Shows inherent limitations of formal systems
- Related to undecidability

Problem-Solving Strategy:

To prove decidable:

- Construct algorithm that always halts
- Show it correctly decides membership

To prove undecidable:

- Reduction from known undecidable problem
- Apply Rice's theorem (if applicable)
- Diagonalization (for fundamental problems)

GATE Tips:

- Halting problem is undecidable (most famous)
- Rice's theorem: Non-trivial properties of RE languages are undecidable
- If L and $\overline{L}$ both RE $\rightarrow$ L is decidable
- Decidable $\subset$ RE $\subset$ All Languages
- Regular $\subset$ CFL: All problems decidable
- CFL: Membership decidable, equivalence undecidable
- TM: Most interesting problems undecidable
- Complement of RE language may not be RE
- RE closed under $\cup$, $\cap$, $\cdot$, * but NOT complement
- Decidable closed under all Boolean operations

Summary Table:

Problem	Regular	CFL	TM
Membership	✓	✓	✗
Emptiness	✓	✓	✗
Finiteness	✓	✓	✗
Equivalence	✓	✗	✗
Containment	✓	✗	✗
Universality	✓	✗	✗
Ambiguity	N/A	✗	✗

(✓ = Decidable, ✗ = Undecidable)

10.6 Dpda (1)

Key Concepts: Deterministic Pushdown Automata (DPDA) are restricted PDAs where at most one transition is possible from any configuration. DPDA languages are proper subset of CFL.

DPDA Definition:

A PDA is **deterministic** if:

1. For each state q and input symbol a, at most one of the following is possible:
   - Transition on (q, a, Z) for some stack symbol Z

- Transition on (q, ε, Z) for some stack symbol Z

2. If ε-transition exists from (q, ε, Z), then no transition on (q, a, Z) for any input symbol a

Formal: For all q, a, Z: $|\delta(q, a, Z)| + |\delta(q, \varepsilon, Z)| \leq 1$

DPDA vs NPDA:

DPDA (Deterministic PDA):

- At most one choice at each step
- Accepts deterministic context-free languages (DCFL)
- Closed under complement
- Not closed under union, intersection, concatenation, Kleene star
- Every DPDA language has unambiguous grammar

NPDA (Nondeterministic PDA):

- Multiple choices possible
- Accepts all context-free languages
- Not closed under complement
- Closed under union, concatenation, Kleene star

Key Theorem: DCFL $\subset$ CFL (proper subset)

Languages:

DCFL Examples:

- $L = \{a^n b^n \mid n \geq 0\}$ (DPDA: push a's, pop on b's)
- $L = \{wcw^R \mid w \in \{a,b\}^*\}$ (c is center marker)
- All regular languages (DFA can be converted to DPDA)
- LR(k) languages (used in compiler design)

CFL but not DCFL:

- $L = \{ww^R \mid w \in \{a,b\}^*\}$ (palindromes without center marker)
- $L = \{a^n b^n \mid n \geq 0\} \cup \{a^n b^{2n} \mid n \geq 0\}$

Proof that palindromes without marker are not DCFL:
DPDA cannot determine center of palindrome without marker, as it would need to guess when to switch from pushing to popping.

Closure Properties:

DCFL is closed under:

- **Complement**: $\bar{L}$ is DCFL if L is DCFL

- **Inverse homomorphism**
- **Intersection with regular language**: L ∩ R is DCFL if L is DCFL and R is regular

DCFL is NOT closed under:

- **Union**: $L_1 \cup L_2$ may not be DCFL
- **Intersection**: $L_1 \cap L_2$ may not be DCFL
- **Concatenation**: $L_1 \cdot L_2$ may not be DCFL
- **Kleene star**: L^* may not be DCFL

Acceptance Modes:

Final State Acceptance: Accept if reach final state after consuming input

Empty Stack Acceptance: Accept if stack becomes empty after consuming input

Key Difference: For DPDA, these two modes are NOT equivalent (unlike NPDA)

- DPDA with final state acceptance ≠ DPDA with empty stack acceptance
- Some DCFL can be accepted by final state but not empty stack

Applications:

1. Compiler Design:

- LR(k) parsers use DPDA
- Deterministic parsing for programming languages
- Efficient O(n) parsing

2. XML/HTML Parsing:

- Balanced tags form DCFL
- Efficient streaming parsers

3. Expression Evaluation:

- Arithmetic expressions with precedence
- Postfix notation evaluation

Decision Problems:

For DCFL:

- **Membership**: Decidable in O(n) time
- **Emptiness**: Decidable
- **Finiteness**: Decidable
- **Equivalence**: Decidable (unlike general CFL!)

GATE Tips:

- DCFL ⊂ CFL (proper subset)
- DCFL closed under complement (CFL is not)
- DCFL not closed under union (unlike regular languages)
- LR(k) languages are DCFL
- Palindromes with center marker: DCFL
- Palindromes without center marker: CFL but not DCFL
- Every DPDA can be converted to equivalent DPDA with single state
- Equivalence decidable for DCFL, undecidable for CFL

10.7 Finite Automata (42)

Key Concepts: Finite automata are computational models with finite memory, fundamental for regular language recognition, lexical analysis, and pattern matching.

Deterministic Finite Automaton (DFA):

Definition: DFA M = (Q, Σ, δ, q_0, F) where:

- Q: Finite set of states
- Σ: Input alphabet
- δ: Q × Σ → Q (transition function)
- q_0 ∈ Q: Start state
- F ⊆ Q: Set of final/accept states

Properties:

- Exactly one transition from each state on each symbol
- Deterministic: no ambiguity in computation
- Memory: Only current state (finite)
- Accepts regular languages

Nondeterministic Finite Automaton (NFA):

Definition: NFA M = (Q, Σ, δ, q_0, F) where:

- δ: Q × Σ → 2^Q (transition to set of states)
- Multiple transitions possible from same state on same symbol
- Accept if ANY computation path leads to final state

ε-NFA (NFA with ε-transitions):

Definition: Like NFA but δ: Q × (Σ ∪ {ε}) → 2^Q

- ε-transitions: Move without consuming input

- More convenient for construction
- Same power as NFA and DFA

Equivalence Theorem: DFA ≡ NFA ≡ ε-NFA ≡ Regular Languages

Subset Construction (NFA to DFA):

Algorithm: Convert NFA to equivalent DFA

1. DFA states = subsets of NFA states
2. Start state = ε-closure of NFA start state
3. $\delta_{DFA}(S, a) = \varepsilon\text{-closure}(\bigcup_{q \in S} \delta_{NFA}(q, a))$
4. Final states = subsets containing NFA final state

ε-closure(q): Set of states reachable from q using only ε-transitions

Example: Convert NFA to DFA

```
NFA: q₀ --a--> {q₀,q₁}, q₁ --b--> {q₂}
DFA states: {q₀}, {q₀,q₁}, {q₂}, ∅
Transitions computed using subset construction
```

Complexity: NFA with n states → DFA with up to 2^n states

DFA Minimization:

Goal: Find DFA with minimum number of states accepting same language

Myhill-Nerode Theorem:
L is regular ⟺ $\equiv_L$ has finite index

where $x \equiv_L y$ iff $\forall z: xz \in L \iff yz \in L$

Minimization Algorithm:

Method 1 - Table Filling:

1. Create table of all state pairs (p,q)
2. Mark pairs where one is final, other is not
3. Mark pair (p,q) if δ(p,a), δ(q,a) already marked for some a
4. Repeat step 3 until no new pairs marked
5. Unmarked pairs are equivalent - merge them

Method 2 - Partition Refinement:

1. Start with partition {F, Q-F} (final vs non-final)
2. Refine partitions: split if states behave differently
3. Two states in same partition iff they have same transitions to same partitions

4. Continue until no more refinement possible
5. Each partition becomes a state in minimal DFA

Properties of Minimal DFA:

- Unique (up to state renaming)
- Minimum number of states for the language
- All states reachable from start state
- No two states equivalent

Regular Expressions to Automata:

Thompson's Construction (Regex to ε-NFA):

- Base cases: ε, ∅, a (single symbol)
- Union: $R_1|R_2$ - create new start state with ε-transitions
- Concatenation: R_1R_2 - connect final states of R_1 to start of R_2
- Kleene star: R* - add ε-transitions for loops and bypass

Result: ε-NFA with O(|r|) states for regex r

Automata to Regular Expressions:

State Elimination Method:

1. Add new start and final states
2. Eliminate states one by one
3. Update transition labels with regular expressions
4. Final regex is label on edge from new start to new final

Arden's Theorem: If $L = AL \cup B$ and $\varepsilon \notin A$, then $L = A^*B$

Applications:

1. Lexical Analysis:

- Tokenize source code
- Keywords, identifiers, numbers, operators
- DFA-based scanners (lex/flex)

2. Pattern Matching:

- String searching algorithms
- Regular expression engines
- Text processing tools (grep, sed)

3. Protocol Verification:

- Model communication protocols
- Verify safety properties
- Network protocol analysis

4. Digital Circuit Design:

- Sequential circuits as finite automata
- State machines in hardware
- Control unit design

5. Bioinformatics:

- DNA sequence analysis
- Pattern recognition in biological data
- Gene finding algorithms

Decision Problems:

For DFA/NFA (all decidable):

- **Membership**: Is $w \in L(M)$? - $O(|w|)$ for DFA
- **Emptiness**: Is $L(M) = \emptyset$? - Check reachability to final states
- **Finiteness**: Is $L(M)$ finite? - Check for cycles in paths to final states
- **Equivalence**: Is $L(M_1) = L(M_2)$? - Minimize and compare
- **Containment**: Is $L(M_1) \subseteq L(M_2)$? - Check $L(M_1) \cap \bar{L}(M_2) = \emptyset$

Pumping Lemma for Regular Languages:

Statement: If L is regular, then $\exists p \geq 1$ such that $\forall w \in L$ with $|w| \geq p$, $\exists x,y,z$ with $w = xyz$ where:

1. $|xy| \leq p$
2. $|y| \geq 1$
3. $\forall k \geq 0$: $xy^k z \in L$

Use: Prove languages are NOT regular

Closure Properties:

Regular languages closed under:

- Union: $L_1 \cup L_2$
- Intersection: $L_1 \cap L_2$
- Complement: $\bar{L}$
- Concatenation: $L_1 \cdot L_2$
- Kleene star: L^*

- Reversal: L^R
- Homomorphism: h(L)
- Inverse homomorphism: $h^{-1}(L)$

Constructions:

- Union/Intersection: Product construction
- Complement: Flip final states (after making DFA complete)
- Concatenation: NFA construction
- Kleene star: Add ε-transitions

State Complexity:

Operations on DFAs with m,n states:

- Union/Intersection: mn states
- Complement: n states (same DFA)
- Concatenation: 2^(m+n-1) states (worst case)
- Kleene star: 2^(n-1) + 1 states

Advanced Topics:

Two-way Finite Automata:

- Can move left or right on input
- Same power as one-way FA (Shepherdson's theorem)
- Useful for certain constructions

Finite Automata with Output:

- **Moore machine**: Output depends on state
- **Mealy machine**: Output depends on state and input
- Used in sequential circuit design

Probabilistic Finite Automata:

- Transitions have probabilities
- Accept with certain probability
- More powerful than regular languages

GATE Tips:

- DFA = NFA = ε-NFA in terms of language recognition power
- Subset construction: NFA to DFA (exponential blowup possible)
- Minimization: Use table filling or partition refinement
- Thompson's construction: Regex to ε-NFA

- Pumping lemma: Prove non-regularity
- Regular languages closed under all Boolean operations
- State complexity: Union/intersection is mn, complement is n
- Every regular language has unique minimal DFA
- Equivalence testing: Minimize both DFAs and compare

Common Constructions:

Example 1: DFA for strings ending in "01"

```
States: q₀ (start), q₁ (saw 0), q₂ (saw 01, final)
Transitions:
q₀ --0--> q₁, q₀ --1--> q₀
q₁ --0--> q₁, q₁ --1--> q₂
q₂ --0--> q₁, q₂ --1--> q₀
```

Example 2: NFA to DFA conversion

```
NFA: Two states, nondeterministic on 'a'
DFA: Four states representing all possible subsets
Exponential blowup demonstrated
```

Example 3: Minimization

```
Original DFA: 5 states
After minimization: 3 states
Equivalent states merged using table filling
```

10.8 Finite State Machines (1)

Key Concepts: Finite State Machines (FSMs) are computational models used to design sequential logic circuits and model systems with discrete states and transitions.

Types of Finite State Machines:

1. Moore Machine:

- Output depends only on current state
- M = (Q, Σ, Δ, δ, λ, q_0) where:
  - Q: States
  - Σ: Input alphabet
  - Δ: Output alphabet
  - δ: Q × Σ → Q (state transition)
  - λ: Q → Δ (output function)

- q_0: Initial state

2. Mealy Machine:

- Output depends on current state AND input
- M = (Q, Σ, Δ, δ, λ, q_0) where:
  - λ: Q × Σ → Δ (output function)

Key Differences:

- **Moore**: Output changes only on state transitions
- **Mealy**: Output can change immediately with input
- **Timing**: Moore has one clock delay, Mealy is immediate
- **Equivalence**: Every Moore machine has equivalent Mealy machine and vice versa

Conversion Between Moore and Mealy:

Moore to Mealy:

- Keep same states and transitions
- Output λ_Mealy(q,a) = λ_Moore(δ(q,a))

Mealy to Moore:

- May need to split states
- Create new states for different outputs
- Can increase number of states

Applications:

1. Digital Circuit Design:

- Sequential logic circuits
- Control units in processors
- State-based controllers

2. Protocol Design:

- Communication protocols
- Network state machines
- Handshaking mechanisms

3. Software Engineering:

- State-based program design
- User interface controllers
- Game state management

4. Hardware Description:

- VHDL/Verilog state machines
- FPGA implementations
- Embedded system controllers

Design Process:

1. **State Identification**: Determine all possible states
2. **Transition Definition**: Define state transitions based on inputs
3. **Output Assignment**: Assign outputs to states (Moore) or transitions (Mealy)
4. **State Encoding**: Assign binary codes to states
5. **Implementation**: Create logic circuits or software

State Minimization:

- Similar to DFA minimization
- Combine equivalent states
- Reduce hardware complexity
- Use implication tables or partitioning

GATE Tips:

- Moore: Output depends only on state
- Mealy: Output depends on state and input
- Moore machines may need more states than equivalent Mealy
- Both types equivalent in computational power
- Used extensively in digital system design
- State minimization reduces implementation cost

(Also see section 6.12 for additional details on automata theory)

10.9 Identify Class Language (31)

Key Concepts: Language classification in Chomsky hierarchy determines computational requirements and decidability properties. Understanding which class a language belongs to is crucial for choosing appropriate parsing/recognition methods.

Chomsky Hierarchy:

Type 3: Regular Languages (REG):

- **Grammar**: Right-linear or left-linear
- **Automaton**: Finite Automaton (DFA/NFA)
- **Memory**: Finite (only current state)
- **Examples**:

- $\{a^n b^n \mid n \geq 0\}$
- Strings ending in "01"
- (a|b)*abb

Type 2: Context-Free Languages (CFL):

- **Grammar**: Context-free ($A \rightarrow \alpha$)
- **Automaton**: Pushdown Automaton (PDA)
- **Memory**: Stack (LIFO)
- **Examples**:
  - $\{a^n b^n \mid n \geq 0\}$
  - Balanced parentheses
  - Palindromes
  - Most programming language constructs

Type 1: Context-Sensitive Languages (CSL):

- **Grammar**: Context-sensitive ($\alpha A \beta \rightarrow \alpha \gamma \beta$, $|\gamma| \geq 1$)
- **Automaton**: Linear Bounded Automaton (LBA)
- **Memory**: Linear in input size
- **Examples**:
  - $\{a^n b^n c^n \mid n \geq 1\}$
  - $\{ww \mid w \in \{a,b\}^*\}$

Type 0: Recursively Enumerable (RE):

- **Grammar**: Unrestricted ($\alpha \rightarrow \beta$)
- **Automaton**: Turing Machine
- **Memory**: Unlimited
- **Examples**:
  - All decidable languages
  - Halting problem (RE but not recursive)

Proper Inclusions: REG $\subset$ CFL $\subset$ CSL $\subset$ RE $\subset$ All Languages

Language Classification Techniques:

1. Pumping Lemmas:

Regular Pumping Lemma:
If L regular, $\exists p$: $\forall w \in L$, $|w| \geq p \Rightarrow w = xyz$ where:

- $|xy| \leq p$, $|y| \geq 1$, $\forall k \geq 0$: $xy^k z \in L$

CFL Pumping Lemma:
If L context-free, $\exists p$: $\forall w \in L$, $|w| \geq p \Rightarrow w = uvxyz$ where:

- $|vxy| \leq p$, $|vy| \geq 1$, $\forall k \geq 0$: $uv^k xy^k z \in L$

2. Closure Properties:

Regular Languages closed under:

- Union, intersection, complement
- Concatenation, Kleene star
- Homomorphism, inverse homomorphism

Context-Free Languages closed under:

- Union, concatenation, Kleene star
- Homomorphism, inverse homomorphism
- NOT closed under: intersection, complement

3. Decision Problems:

Regular: All standard problems decidable
CFL: Membership, emptiness decidable; equivalence undecidable
CSL: Membership decidable; emptiness undecidable
RE: Only membership semi-decidable

Classification Examples:

Example 1: $L = \{a^n b^n c^n \mid n \geq 0\}$

- **Not Regular**: Use pumping lemma (can't count three symbols)
- **Not CFL**: Use CFL pumping lemma
- **Is CSL**: Grammar $S \rightarrow aSbc \mid \varepsilon$ with length-increasing productions
- **Classification**: CSL but not CFL

Example 2: $L = \{ww \mid w \in \{a,b\}^*\}$

- **Not CFL**: Use CFL pumping lemma
- **Is CSL**: Can be recognized by LBA
- **Classification**: CSL but not CFL

Example 3: $L = \{a^n \mid n \text{ is prime}\}$

- **Not Regular**: Infinite number of "periods"
- **Not CFL**: No stack-based recognition
- **Is CSL**: Can check primality with linear space
- **Classification**: CSL but not CFL

Example 4: L = {a^n | n = 2^m for some m}

- **Not Regular**: Use pumping lemma
- **Not CFL**: Exponential growth pattern
- **Is CSL**: Can be recognized by LBA
- **Classification**: CSL but not CFL

Practical Classification Strategy:

Step 1: Check if Regular

- Can you build DFA/NFA?
- Does it satisfy regular pumping lemma?
- Is it closed under regular operations?

Step 2: Check if Context-Free

- Can you build PDA?
- Can you write CFG?
- Does it satisfy CFL pumping lemma?
- Check closure properties

Step 3: Check if Context-Sensitive

- Can you write CSG?
- Can you build LBA?
- Is membership decidable?

Step 4: Check if Recursively Enumerable

- Can you build TM that accepts it?
- Is it semi-decidable?

Common Non-Regular Languages:

- {a^nb^n | n ≥ 0} - CFL
- {ww^R | w ∈ {a,b}*} - CFL (palindromes)
- {a^n | n is perfect square} - Not regular

Common Non-CFL Languages:

- {$a^nb^nc^n$ | n ≥ 0} - CSL
- {ww | w ∈ {a,b}*} - CSL
- {$a^nb^nc^n$ | 0 ≤ n ≤ m ≤ k} - CSL

Intersection Properties:

- CFL ∩ Regular = CFL
- CFL ∩ CFL may not be CFL
- CSL ∩ CSL = CSL

GATE Tips:

- Use pumping lemmas to prove non-membership in class
- Check closure properties to determine class boundaries
- Regular ⊂ CFL ⊂ CSL ⊂ RE (proper inclusions)
- Most programming languages are CFL (with some CSL features)
- $\{a^n b^n c^n \mid n \geq 0\}$ is classic CSL example
- $\{ww \mid w \in \Sigma^*\}$ is CSL but not CFL
- Complement of CFL may not be CFL
- Intersection of two CFLs may not be CFL
- All finite languages are regular
- Unary languages (over single symbol) have special properties

Decision Procedure:

1. **Finite language** ⇒ Regular
2. **Can count with DFA** ⇒ Regular
3. **Need stack for matching** ⇒ CFL
4. **Need to compare distant parts** ⇒ CSL
5. **Undecidable properties** ⇒ RE but not recursive

Applications:

- **Compiler Design**: Language classification determines parsing method
- **Formal Verification**: Model checking complexity
- **Database Theory**: Query language expressiveness
- **Bioinformatics**: DNA sequence pattern complexity

10.10 Minimal State Automata (25)

Key Concepts: Minimal state automata have the fewest states possible while recognizing the same language. Based on Myhill-Nerode theorem and equivalence classes.

Myhill-Nerode Theorem:

Definition: For language L, define equivalence relation $\equiv_L$:
$x \equiv_L y$ iff $\forall z \in \Sigma^*: xz \in L \Leftrightarrow yz \in L$

Theorem: The following are equivalent:

1. L is regular

2. ≡_L has finite index (finite number of equivalence classes)
3. L is recognized by some DFA

Corollary: If L is regular, then the minimal DFA for L has exactly index(≡_L) states.

Equivalence Classes:

Right Invariant: ≡_L is right invariant:
If x ≡_L y, then xz ≡_L yz for all z

Distinguishing String: String z distinguishes x and y if exactly one of xz, yz is in L

Example: L = {strings ending in 'a'}

- [""]: strings ending in 'b' or empty
- ["a"]: strings ending in 'a'
- Two equivalence classes ⇒ minimal DFA has 2 states

DFA Minimization Algorithms:

Algorithm 1: Table Filling (Pair Marking):

```
1. Create table of all state pairs (p,q) where p ≠ q
2. Mark pairs (p,q) where p ∈ F and q ∉ F (or vice versa)
3. For each unmarked pair (p,q):
   For each symbol a ∈ Σ:
     If (δ(p,a), δ(q,a)) is marked, mark (p,q)
4. Repeat step 3 until no new pairs marked
5. Unmarked pairs are equivalent - merge them
```

Time Complexity: O(n²|Σ|)

Algorithm 2: Partition Refinement:

```
1. Initial partition: P = {F, Q \ F}
2. For each partition block B in P:
   For each symbol a:
     Split B based on which partition δ(q,a) belongs to
3. Repeat until no more refinement possible
4. Each final partition block becomes a state
```

Properties of Minimal DFA:

1. **Uniqueness**: Minimal DFA is unique up to isomorphism
2. **Reachability**: All states reachable from start state
3. **Distinguishability**: No two states are equivalent
4. **Optimality**: Fewest possible states for the language

Constructing Minimal DFA from Myhill-Nerode Classes:

1. **States**: Q = {Σ*/≡_L} (equivalence classes)
2. **Start state**: [""]≡_L (class of empty string)
3. **Transitions**: δ([x]≡_L, a) = [xa]≡_L
4. **Final states**: F = {[x]≡_L : x ∈ L}

Examples:

Example 1: L = (a|b)*a (strings ending in 'a')

- Equivalence classes:
  - [""]: strings not ending in 'a' (including "")
  - ["a"]: strings ending in 'a'
- Minimal DFA: 2 states

Example 2: L = {a^n | n ≡ 0 (mod 3)}

- Equivalence classes:
  - [""]: strings with length ≡ 0 (mod 3)
  - ["a"]: strings with length ≡ 1 (mod 3)
  - ["aa"]: strings with length ≡ 2 (mod 3)
- Minimal DFA: 3 states

Example 3: L = {w | w contains substring "ab"}

- Equivalence classes:
  - [""]: no 'a' seen, or last char not 'a'
  - ["a"]: ends with 'a', no "ab" seen
  - ["ab"]: "ab" substring seen
- Minimal DFA: 3 states

Minimization Process Example:

Original DFA: 5 states {q_0, q_1, q_2, q_3, q_4}
Final states: {q_2, q_4}

Table Filling:

```
Mark (q₀,q₂), (q₀,q₄), (q₁,q₂), (q₁,q₄), (q₃,q₂), (q₃,q₄)
Check transitions and mark additional pairs
Unmarked pairs: (q₀,q₁), (q₂,q₄)
Merge: {q₀,q₁}, {q₂,q₄}, {q₃}
```

Result: 3-state minimal DFA

Applications:

1. Compiler Design:

- Minimize lexical analyzer DFAs
- Reduce memory and improve performance
- Optimize regular expression engines

2. Hardware Design:

- Minimize state machines in digital circuits
- Reduce flip-flops and logic gates
- Lower power consumption

3. Protocol Verification:

- Simplify protocol state machines
- Reduce verification complexity
- Find equivalent protocol states

4. Pattern Matching:

- Optimize string matching automata
- Reduce space complexity
- Improve search performance

Advanced Topics:

Incremental Minimization:

- Add states one by one
- Maintain minimality during construction
- Useful for online algorithms

Minimization with Don't Cares:

- Some transitions undefined
- More opportunities for state merging
- Used in circuit optimization

Approximate Minimization:

- Trade accuracy for smaller automata
- Useful in machine learning applications
- Probabilistic automata minimization

Complexity Results:

Time Complexity:

- Table filling: $O(n^2|\Sigma|)$
- Partition refinement: $O(n|\Sigma| \log n)$
- Hopcroft's algorithm: $O(n|\Sigma| \log n)$ (optimal)

Space Complexity: $O(n^2)$ for table filling

Lower Bounds: $\Omega(n \log n)$ for comparison-based algorithms

GATE Tips:

- Myhill-Nerode: $x \equiv_L y$ iff $\forall z: xz \in L \Leftrightarrow yz \in L$
- Number of equivalence classes = number of states in minimal DFA
- Table filling algorithm: mark distinguishable pairs
- Partition refinement: split based on transition behavior
- Minimal DFA is unique (up to state renaming)
- All states in minimal DFA are reachable and distinguishable
- Minimization reduces both space and time complexity
- Regular language is finite iff its minimal DFA has no cycles

Common Mistakes:

- Forgetting to check reachability before minimization
- Not considering all symbols when marking pairs
- Confusing equivalence with similarity
- Assuming minimization always reduces states significantly

Verification:

- Check that minimized DFA accepts same language
- Verify all states are reachable
- Confirm no two states are equivalent
- Test with representative strings from each equivalence class

10.11 Non Determinism (6)

Key Concepts: Nondeterminism allows multiple possible transitions from a state, providing computational flexibility at the cost of exponential state explosion when converted to deterministic form.

Nondeterministic Finite Automaton (NFA):

Definition: NFA $M = (Q, \Sigma, \delta, q_0, F)$ where:

- Q: Finite set of states

- Σ: Input alphabet
- δ: Q × Σ → 2^Q (transition to set of states)
- $q_0 \in Q$: Start state
- $F \subseteq Q$: Set of final states

Key Properties:

- Multiple transitions possible from same state on same input
- ε-transitions allowed (move without consuming input)
- Accept if ANY computation path reaches final state
- Guess-and-verify paradigm

NFA vs DFA Power:

Theorem: NFAs and DFAs recognize exactly the same class of languages (regular languages)

Proof Idea:

- DFA ⇒ NFA: Trivial (DFA is special case of NFA)
- NFA ⇒ DFA: Subset construction algorithm

Subset Construction Algorithm:

Goal: Convert NFA to equivalent DFA

Algorithm:

```
1. DFA states = subsets of NFA states (2^Q)
2. Start state = ε-closure({q₀})
3. For each DFA state S and symbol a:
   δ_DFA(S, a) = ε-closure(∪_{q∈S} δ_NFA(q, a))
4. Final states = subsets containing at least one NFA final state
5. Remove unreachable states
```

ε-closure(S): Set of states reachable from S using only ε-transitions

State Explosion:

- NFA with n states ⇒ DFA with up to 2^n states
- Exponential blowup in worst case
- Many practical cases have polynomial blowup

Example: NFA to DFA Conversion

NFA: Accepts strings ending in "01"

```
States: {q₀, q₁, q₂}
Transitions:
q₀ --0--> {q₀, q₁}
q₀ --1--> {q₀}
q₁ --1--> {q₂}
Final: {q₂}
```

DFA (after subset construction):

```
States: {∅, {q₀}, {q₀,q₁}, {q₀,q₂}}
Transitions computed using subset construction
Final: {{q₀,q₂}}
```

Advantages of Nondeterminism:

1. Conceptual Simplicity:

- Easier to design for some languages
- Natural "guess and verify" approach
- More intuitive for certain patterns

2. Compact Representation:

- Fewer states than equivalent DFA
- Exponentially smaller in some cases
- Better for theoretical analysis

3. Compositional Operations:

- Union: Add new start state with ε-transitions
- Concatenation: Connect final states to start states
- Kleene star: Add ε-loops

4. Regular Expression Construction:

- Thompson's construction naturally produces NFAs
- Direct translation from regex to NFA
- Avoids intermediate exponential blowup

Disadvantages of Nondeterminism:

1. Implementation Complexity:

- Need to track multiple active states
- Backtracking or parallel simulation required
- More complex than DFA simulation

2. Time Complexity:

- NFA simulation: O(n²) or O(n³) depending on method
- DFA simulation: O(n)
- Space-time tradeoff

3. Exponential Conversion:

- Converting to DFA can be exponential
- May not be practical for large NFAs
- Memory limitations

NFA Simulation Algorithms:

Method 1: Subset Tracking:

```
SimulateNFA(NFA, input):
    current = ε-closure({start_state})
    for each symbol a in input:
        next = ∅
        for each state q in current:
            next = next ∪ δ(q, a)
        current = ε-closure(next)
    return current ∩ F ≠ ∅
```

Method 2: Recursive Backtracking:

```
Accept(state, remaining_input):
    if remaining_input is empty:
        return state ∈ F
    for each transition from state on first symbol:
        if Accept(next_state, rest_of_input):
            return true
    return false
```

ε-NFA (NFA with ε-transitions):

Extended Definition: δ: Q × (Σ ∪ {ε}) → 2^Q

ε-transitions:

- Move without consuming input
- Useful for construction algorithms
- Can create "spontaneous" state changes

ε-closure Computation:

```
EpsilonClosure(states):
    closure = states
    stack = states
    while stack not empty:
        q = pop(stack)
        for each r in δ(q, ε):
            if r not in closure:
                add r to closure
                push r onto stack
    return closure
```

Applications of Nondeterminism:

1. Regular Expression Engines:

- Thompson's construction: regex → ε-NFA
- Efficient for pattern matching
- Backtracking implementations

2. Lexical Analysis:

- Token recognition with multiple patterns
- Longest match disambiguation
- Efficient scanner generation

3. Model Checking:

- Nondeterministic system models
- Property verification
- State space exploration

4. Parsing Theory:

- Nondeterministic parsers
- Ambiguity handling
- Parse forest generation

Theoretical Implications:

Complexity Theory:

- P vs NP related to determinism vs nondeterminism
- Polynomial time vs nondeterministic polynomial time
- Space complexity: NSPACE vs DSPACE

Savitch's Theorem: NSPACE(s(n)) ⊆ DSPACE(s(n)²)

Immerman-Szelepcsényi Theorem: NSPACE(s(n)) = co-NSPACE(s(n))

Practical Considerations:

When to Use NFA:

- Conceptual design phase
- Theoretical analysis
- Space-critical applications
- Compositional constructions

When to Use DFA:

- Implementation phase
- Performance-critical applications
- Simple simulation required
- Hardware implementations

Hybrid Approaches:

- Lazy DFA construction (build states on demand)
- NFA simulation with memoization
- Compressed DFA representations

GATE Tips:

- NFA and DFA have same computational power
- Subset construction: 2^n possible DFA states from n NFA states
- ε-closure: states reachable via ε-transitions only
- NFA accepts if ANY path leads to final state
- DFA accepts if THE path leads to final state
- Thompson's construction: regex to ε-NFA
- NFA simulation: track set of active states
- Exponential blowup is worst-case, not always realized
- Regular operations easier to implement with NFAs

Common Examples:

Example 1: NFA for (a|b)*abb

- Guess when "abb" starts
- Nondeterministically choose to start matching
- Much simpler than equivalent DFA

Example 2: Union of two NFAs

- Create new start state
- Add ε-transitions to both original start states
- No state explosion during construction

Example 3: Exponential blowup

- NFA: (a|b)*a(a|b)^{n-1}
- DFA needs 2^n states to remember last n symbols
- Demonstrates worst-case complexity

10.12 Number of States (1)

Key Concepts: The number of states required for automata depends on the language complexity and construction method. Understanding state bounds is crucial for implementation efficiency.

State Complexity for Regular Expressions:

Thompson's Construction (Regex to ε-NFA):

- **Linear size**: ε-NFA has O(|r|) states for regex r
- **Compositional**: Each operator adds constant states
- **Base cases**:
  - ε: 2 states
  - ∅: 2 states
  - a: 2 states

Construction Rules:

- **Union ($r_1|r_2$)**: Add 2 states + states for r_1, r_2
- **Concatenation (r_1r_2)**: Merge final of r_1 with start of r_2
- **Kleene star (r*)**: Add 2 states + states for r

Total states: At most 2|r| states for regex r

Subset Construction (ε-NFA to DFA):

- **Exponential blowup**: Up to 2^n DFA states from n NFA states
- **Worst case**: Often not realized in practice
- **Average case**: Usually polynomial growth

State Bounds for Common Patterns:

1. Simple Patterns:

- a*: 1 state (DFA)
- (a|b)*: 1 state (DFA)

- a*b*: 2 states (DFA)

2. Counting Patterns:

- a^n: n+1 states (DFA)
- (a|b)^n: 2^n states (DFA worst case)
- Strings with exactly n a's: n+1 states (DFA)

3. Suffix Patterns:

- .*abc: 4 states (DFA)
- .*(abc|def): 7 states (DFA)
- .*a.{n}b: n+3 states (DFA)

4. Complex Patterns:

- (a|b)*a(a|b)^{n-1}: 2^n states (DFA)
- Patterns requiring memory of last n symbols: 2^n states

Minimization Impact:

Before Minimization: Construction may produce redundant states
After Minimization: Optimal number of states for the language

Myhill-Nerode Bound: Minimal DFA has exactly as many states as equivalence classes in ≡_L

Practical State Estimation:

Factors Affecting State Count:

1. **Pattern complexity**: More complex patterns need more states
2. **Lookahead requirements**: Patterns needing future context
3. **Memory requirements**: How much history must be remembered
4. **Ambiguity**: Nondeterministic choices increase states

Estimation Heuristics:

- **Linear patterns**: O(length) states
- **Alternation**: Sum of component states
- **Repetition**: May not increase states significantly
- **Lookahead**: Exponential in lookahead distance

Examples with State Analysis:

Example 1: Regex (a|b)*abb

- **Thompson NFA**: ~8 states

- **Subset construction**: Up to 2^8 = 256 states
- **Actual DFA**: 4 states
- **Minimal DFA**: 4 states

Example 2: Regex a*b*c*

- **Thompson NFA**: ~6 states
- **Direct DFA**: 4 states
- **Minimal DFA**: 4 states

Example 3: Regex (a|b)*a(a|b)^2

- **Thompson NFA**: ~10 states
- **DFA**: 8 states (needs to remember last 3 symbols)
- **Minimal DFA**: 8 states

State Optimization Techniques:

1. Direct Construction:

- Build DFA directly from regex
- Avoid intermediate NFA
- Tools: Brzozowski derivatives

2. Lazy Construction:

- Build DFA states on demand
- Only create reachable states
- Useful for large theoretical state spaces

3. Compressed Representations:

- Share common state structures
- Use decision diagrams (BDDs)
- Exploit pattern regularities

Applications:

1. Regular Expression Engines:

- Estimate memory requirements
- Choose between NFA and DFA simulation
- Optimize pattern compilation

2. Lexical Analyzers:

- Size scanner tables

- Memory allocation for state machines
- Performance prediction

3. Network Security:

- Intrusion detection pattern matching
- Deep packet inspection
- Firewall rule compilation

4. Bioinformatics:

- DNA sequence pattern matching
- Protein structure analysis
- Phylogenetic tree construction

Complexity Results:

Lower Bounds:

- Some languages require exponential states
- No general polynomial bound for regex to DFA
- Specific patterns have known lower bounds

Upper Bounds:

- Thompson + subset construction: 2^{O(|r|)}
- Direct methods: Often better in practice
- Specialized algorithms for restricted classes

Trade-offs:

Space vs Time:

- **NFA**: Less space, more time per input symbol
- **DFA**: More space, constant time per input symbol
- **Hybrid**: Lazy DFA construction

Construction vs Runtime:

- **Precompute DFA**: High construction cost, fast runtime
- **NFA simulation**: Low construction cost, slower runtime
- **JIT compilation**: Compile patterns on first use

GATE Tips:

- Thompson construction: O(|r|) states for regex r
- Subset construction: Up to 2^n states from n-state NFA

- Minimal DFA: Unique and optimal for the language
- State count depends on pattern complexity and memory requirements
- Exponential blowup is worst-case, not typical
- Direct DFA construction often better than NFA conversion
- Minimization can significantly reduce state count
- Practical implementations use optimizations to reduce states

Common Patterns and State Counts:

Pattern	NFA States	DFA States	Minimal DFA
a*	2	1	1
(a|b)*	4	1	1
a*b*	4	2	2
(a|b)*abb	8	4	4
a^n	2n	n+1	n+1
(a|b)^n	2n	2^n	2^n

Memory Estimation:

- Each state: ~10-100 bytes (depending on implementation)
- Transition table: states × alphabet size × pointer size
- Total memory: O(states × alphabet size)
- Large DFAs may require disk storage or compression

10.13 Pumping Lemma (2)

Key Concepts: Pumping lemmas are fundamental tools for proving that languages are NOT in certain classes. They exploit the finite memory limitations of automata.

Pumping Lemma for Regular Languages:

Statement: If L is regular, then ∃ pumping length p ≥ 1 such that:
∀ string w ∈ L with |w| ≥ p, ∃ strings u, v, x such that:

1. w = uvx
2. |uv| ≤ p (pump early)
3. |v| ≥ 1 (pump something non-empty)
4. ∀ k ≥ 0: uv^k x ∈ L (pumping preserves membership)

Intuition:

- DFA has finite states (p states)
- String of length ≥ p must revisit some state

- Loop between repeated state can be pumped

Proof:
Let M be DFA with p states recognizing L.
For w = $a_1a_2...a_n$ with n ≥ p:

- Computation visits states $q_0, q_1, ..., q_n$
- By pigeonhole principle: some state repeats in first p+1 positions
- Let q_i = q_j where 0 ≤ i < j ≤ p
- Set u = $a_1...a_i$, $v = a_{i+1}...a_j$, $x = a_{j+1}...a_n$
- Loop from q_i to q_j can be repeated any number of times

Using Pumping Lemma to Prove Non-Regularity:

Method (Proof by contradiction):

1. Assume L is regular
2. Let p be the pumping length guaranteed by lemma
3. Choose string w ∈ L with |w| ≥ p (strategic choice!)
4. For ALL possible divisions w = uvx satisfying conditions 1-3
5. Show ∃ k such that uv^k x ∉ L (contradiction)
6. Therefore L is not regular

Strategic String Selection:

- Use p in string definition to ensure |w| ≥ p
- Choose w that makes pumping impossible
- Common patterns: a^p b^p, a^{p!}, a^{p^2}

Classic Examples:

Example 1: L = {a^n b^n | n ≥ 0}

Proof:

1. Assume L is regular with pumping length p
2. Choose w = a^p b^p ∈ L, |w| = 2p ≥ p
3. Any division w = uvx with |uv| ≤ p and |v| ≥ 1:
   - v consists only of a's (since |uv| ≤ p)
   - v = a^j for some 1 ≤ j ≤ p
4. Pump k = 2: uv^2 x = a^{p+j} b^p
5. Since j ≥ 1: p+j > p, so uv^2 x ∉ L
6. Contradiction! L is not regular.

Example 2: L = {ww | w ∈ {a,b}*}

Proof:

1. Assume L is regular with pumping length p
2. Choose w = a^p b a^p b ∈ L, |w| = 2p+2 ≥ p
3. Any division with |uv| ≤ p:
   - v is substring of first a^p
   - v = a^j for some 1 ≤ j ≤ p
4. Pump k = 2: uv^2 x = a^{p+j} b a^p b
5. First half: a^{p+j} b, second half: a^p b
6. Since j ≥ 1: first ≠ second, so uv^2 x ∉ L
7. Contradiction! L is not regular.

Example 3: L = {a^{n^2} | n ≥ 0}

Proof:

1. Assume L is regular with pumping length p
2. Choose w = a^{p^2} ∈ L, |w| = p^2 ≥ p
3. Any division with |uv| ≤ p and |v| ≥ 1:
   - v = a^j for some 1 ≤ j ≤ p
4. Pump k = 2: |uv^2 x| = p^2 + j
5. Need to show p^2 < p^2 + j < (p+1)^2 = p^2 + 2p + 1
6. Since 1 ≤ j ≤ p < 2p + 1: p^2 + j is not perfect square
7. So uv^2 x ∉ L. Contradiction!

Pumping Lemma for Context-Free Languages:

Statement: If L is context-free, then ∃ pumping length p such that:
∀ string w ∈ L with |w| ≥ p, ∃ strings u,v,x,y,z such that:

1. w = uvxyz
2. |vxy| ≤ p (pump within bounded region)
3. |vy| ≥ 1 (pump something non-empty)
4. ∀ k ≥ 0: uv^k xy^k z ∈ L (synchronized pumping)

Key Difference: Two substrings (v and y) pumped simultaneously

Example: L = {a^n b^n c^n | n ≥ 0} is not context-free

Proof:

1. Choose w = a^p b^p c^p
2. Any division with |vxy| ≤ p:
   - vxy spans at most 2 of the 3 symbol types
   - Pumping changes counts of at most 2 symbol types

3. But need all 3 counts to increase equally
4. Contradiction!

Common Mistakes:

1. Wrong Direction:

- Pumping lemma proves non-regularity only
- Cannot prove regularity using pumping lemma

2. Insufficient Cases:

- Must consider ALL possible divisions
- Not just one convenient division

3. Wrong String Choice:

- String must depend on p
- Must be in the language
- Should make pumping impossible

4. Incorrect Pumping:

- Must show pumping fails for some k
- k = 0 (deletion) and k = 2 (duplication) most common

Advanced Applications:

Ogden's Lemma: Stronger version of CFL pumping lemma

- Mark certain positions as "distinguished"
- More powerful for proving non-context-freeness

Interchange Lemma: For regular languages

- If strings x, y have same effect, can interchange them
- Useful when pumping lemma doesn't apply directly

GATE Tips:

- Pumping lemma: proof by contradiction only
- Choose string w strategically (use p in definition)
- Consider ALL possible divisions satisfying conditions
- Show pumping fails for some k (usually k = 0 or k = 2)
- Regular pumping: one substring pumped
- CFL pumping: two substrings pumped simultaneously
- Cannot prove language IS regular using pumping lemma

- Must show pumping fails for ALL valid divisions

Problem-Solving Strategy:

1. **Assume** language is regular (for contradiction)
2. **Let** p be pumping length
3. **Choose** w ∈ L with |w| ≥ p (use p in definition)
4. **Consider** all divisions w = uvx with |uv| ≤ p, |v| ≥ 1
5. **Show** for some k: uv^k x ∉ L
6. **Conclude** contradiction, so language not regular

Template for GATE Answers:

```
Proof by contradiction using pumping lemma:
1. Assume L is regular with pumping length p
2. Choose w = [string using p] ∈ L, |w| ≥ p
3. By pumping lemma, w = uvx where:
   - |uv| ≤ p
   - |v| ≥ 1
   - ∀k ≥ 0: uv^k x ∈ L
4. Since |uv| ≤ p, v must be [analyze structure]
5. Consider uv^k x for k = [choose value]:
   [Show this string is not in L]
6. Contradiction! Therefore L is not regular.
```

10.14 Pushdown Automata (15)

Key Concepts: Pushdown Automata (PDA) extend finite automata with a stack, providing the computational power needed to recognize context-free languages.

Pushdown Automaton Definition:

PDA: M = (Q, Σ, Γ, δ, q_0, Z_0, F) where:

- Q: Finite set of states
- Σ: Input alphabet
- Γ: Stack alphabet
- δ: Q × (Σ ∪ {ε}) × Γ → 2^{Q × Γ*} (transition function)
- q_0 ∈ Q: Start state
- Z_0 ∈ Γ: Initial stack symbol
- F ⊆ Q: Final states

Transition: δ(q, a, X) = {(p_1, α_1), (p_2, α_2), ...}

- From state q, reading input a, with X on top of stack
- Go to state p_i and replace X with string α_i

Configuration: (q, w, γ) where:

- q: current state
- w: remaining input
- γ: stack contents (top at left)

Computation: Sequence of configurations connected by transitions

Acceptance Modes:

1. Acceptance by Final State:

- Accept if reach final state after consuming all input
- Stack contents irrelevant
- L(M) = {w | $(q_0, w, Z_0) \vdash$ *(f, ε, γ) for some f ∈ F, γ ∈ Γ*}

2. Acceptance by Empty Stack:

- Accept if stack becomes empty after consuming all input
- Current state irrelevant
- N(M) = {w | $(q_0, w, Z_0) \vdash^*$ (q, ε, ε) for some q ∈ Q}

Equivalence: Both acceptance modes recognize exactly the context-free languages

Conversion Between Modes:

Final State to Empty Stack:

1. Add new start state with ε-transition pushing new bottom marker
2. From each final state, add ε-transitions to pop entire stack
3. Accept when stack becomes empty

Empty Stack to Final State:

1. Add new bottom marker to stack
2. Add new final state
3. When original stack becomes empty (only marker remains), transition to final state

PDA Examples:

Example 1: L = {a^n b^n | n ≥ 0}

```
States: {q₀, q₁, q₂}
Transitions:
δ(q₀, a, Z₀) = {(q₀, AZ₀)}  // Push A for each a
δ(q₀, a, A) = {(q₀, AA)}      // Continue pushing A's
δ(q₀, b, A) = {(q₁, ε)}       // Start popping on first b
δ(q₁, b, A) = {(q₁, ε)}       // Continue popping A's
```

```
δ(q₁, ε, Z₀) = {(q₂, Z₀)}     // Accept when done
Final states: {q₂}
```

Example 2: L = {ww^R | w ∈ {a,b}*} (palindromes with center)

```
Nondeterministically guess center of palindrome:
- Push symbols onto stack
- At guessed center, switch to popping mode
- Pop and match remaining input
- Accept if stack empty and input consumed
```

Example 3: Balanced Parentheses

```
L = {w | w has balanced parentheses}
Push '(' onto stack
Pop '(' when seeing ')'
Accept if stack empty at end
```

Deterministic vs Nondeterministic PDA:

NPDA (Nondeterministic PDA):

- Multiple transitions possible
- Accepts all context-free languages
- More powerful than DPDA

DPDA (Deterministic PDA):

- At most one transition possible from each configuration
- Accepts deterministic context-free languages (DCFL)
- DCFL ⊂ CFL (proper subset)

Key Difference: DPDA ≠ NPDA in power (unlike DFA = NFA)

PDA Construction from CFG:

Algorithm: Convert CFG to PDA

1. **Single state**: PDA has only one state
2. **Stack simulation**: Stack contains sentential forms
3. **Transitions**:
   - For terminal a: δ(q, a, a) = {(q, ε)} (match and pop)
   - For production A → α: δ(q, ε, A) = {(q, α)} (expand nonterminal)
4. **Start**: Push start symbol onto stack
5. **Accept**: When stack empty and input consumed

Example: CFG S → aSb | ε

```
PDA transitions:
δ(q, ε, S) = {(q, aSb), (q, ε)}  // Productions
δ(q, a, a) = {(q, ε)}              // Match terminal a
δ(q, b, b) = {(q, ε)}              // Match terminal b
```

CFG Construction from PDA:

Algorithm: Convert PDA to CFG

1. **Variables**: [q, A, p] represents "from state q with A on stack, reach state p with A popped"
2. **Productions**: Based on PDA transitions
3. **Start symbol**: [q_0, Z_0, q] for each state q
4. **Complex construction**: Handles all possible stack behaviors

Closure Properties:

CFL is closed under:

- Union: $L_1 \cup L_2$
- Concatenation: $L_1 \cdot L_2$
- Kleene star: L*
- Homomorphism: h(L)
- Inverse homomorphism: h^{-1}(L)

CFL is NOT closed under:

- Intersection: $L_1 \cap L_2$ (may not be CFL)
- Complement: $\overline{L}$ (may not be CFL)

But: CFL ∩ Regular = CFL (intersection with regular language)

Decision Problems for PDA:

Decidable:

- **Membership**: Is $w \in L(M)$? (CYK algorithm for CFG)
- **Emptiness**: Is $L(M) = \emptyset$?
- **Finiteness**: Is L(M) finite?

Undecidable:

- **Equivalence**: Is $L(M_1) = L(M_2)$?
- **Containment**: Is $L(M_1) \subseteq L(M_2)$?
- **Intersection emptiness**: Is $L(M_1) \cap L(M_2) = \emptyset$?
- **Universality**: Is $L(M) = \Sigma^*$?

- **Ambiguity**: Is CFG ambiguous?

Applications:

1. Parsing:

- Syntax analysis in compilers
- LL and LR parsers based on DPDA
- Parse tree construction

2. Expression Evaluation:

- Arithmetic expressions with precedence
- Balanced parentheses checking
- Postfix notation conversion

3. XML/HTML Processing:

- Nested tag matching
- Document structure validation
- Tree-structured data parsing

4. Programming Languages:

- Block structure (begin-end, {-})
- Procedure call/return matching
- Scope resolution

Advanced Topics:

Real-time PDA: Must make move on each input symbol
Counter Automata: PDA with counters instead of stack
Visibly Pushdown Automata: Input determines stack operations
Higher-order PDA: Stack of stacks (more powerful)

Implementation Considerations:

Stack Representation:

- Array with top pointer
- Linked list
- Implicit recursion stack

Efficiency:

- DPDA: O(n) time for input length n
- NPDA: Exponential time in worst case

- Early termination on acceptance

Memory Management:

- Stack overflow detection
- Garbage collection for unused stack symbols
- Bounded stack approximations

GATE Tips:

- PDA = FA + Stack (LIFO memory)
- PDA recognizes exactly the context-free languages
- DPDA ⊂ NPDA in power (unlike FA)
- Two acceptance modes: final state and empty stack
- CFG ↔ PDA conversion algorithms exist
- CFL closed under union, concatenation, star
- CFL NOT closed under intersection, complement
- Membership decidable, equivalence undecidable
- Stack operations: push, pop, peek
- ε-transitions allowed (no input consumed)

Common PDA Patterns:

1. Counting Pattern: a^n b^n

- Push for a's, pop for b's
- Accept when counts match

2. Palindrome Pattern: ww^R

- Push first half, pop second half
- Nondeterministically guess center

3. Balanced Pattern: Parentheses

- Push opening symbols
- Pop on closing symbols
- Accept when balanced

4. Nested Pattern: Nested structures

- Stack tracks nesting depth
- Push on entry, pop on exit

Debugging PDA:

- Trace configurations step by step

- Check stack contents at each step
- Verify acceptance conditions
- Consider all nondeterministic paths

10.15 Recursive and Recursively Enumerable Languages (16)

Key Concepts: The hierarchy of recursive and recursively enumerable languages defines the boundaries of computability and decidability in theoretical computer science.

Definitions:

Recursively Enumerable (RE) Language:

- L is RE if ∃ Turing Machine M such that:
  - If w ∈ L, then M(w) halts and accepts
  - If w ∉ L, then M(w) either halts and rejects OR loops forever
- Also called **semi-decidable** or **Turing-recognizable**
- M may not halt on strings not in L

Recursive Language (Decidable):

- L is recursive if ∃ Turing Machine M such that:
  - If w ∈ L, then M(w) halts and accepts
  - If w ∉ L, then M(w) halts and rejects
- Also called **decidable** or **Turing-decidable**
- M always halts (total function)

Key Relationship: Recursive ⊂ RE ⊂ All Languages

Characterizations:

RE Languages:

1. **TM Recognition**: Accepted by some Turing Machine
2. **Enumeration**: Can be enumerated by TM (possibly with repetitions)
3. **Semi-decision**: Membership is semi-decidable
4. **Projection**: Projection of recursive relation

Recursive Languages:

1. **TM Decision**: Decided by some Turing Machine
2. **Complement**: Both L and $\overline{L}$ are RE
3. **Total computation**: Always terminates
4. **Effective procedure**: Algorithm exists

Fundamental Theorem:
L is recursive ⟺ L and ¬L are both RE

Proof (⇒):
If L recursive, then TM M decides L.
Construct M_1 for L: simulate M, accept if M accepts
Construct M_2 for ¬L: simulate M, accept if M rejects
Both M_1, M_2 always halt, so L and ¬L are RE.

Proof (⇐):
If L and ¬L are RE with TMs M_1, M_2:
Construct M: simulate M_1 and M_2 in parallel

- If M_1 accepts, accept
- If M_2 accepts, reject
  Since every string is in L or ¬L, one machine must halt.
  Therefore M always halts and decides L.

Examples:

Recursive Languages:

- All context-free languages
- All regular languages
- {w | w is valid C program}
- {(M, w) | TM M halts on input w in ≤ 1000 steps}
- Arithmetic truths (Presburger arithmetic)

RE but not Recursive:

- **Halting Problem**: L_H = {(M, w) | TM M halts on input w}
- **Acceptance Problem**: L_A = {(M, w) | TM M accepts input w}
- **Blank Tape Halting**: {M | TM M halts on empty input}
- **Totality Problem**: {M | TM M halts on all inputs}

Not RE:

- **Complement of Halting Problem**: ¬L_H
- **Non-halting Problem**: {(M, w) | TM M doesn't halt on w}
- **Complement of any RE-but-not-recursive language**

Closure Properties:

RE Languages closed under:

- **Union**: $L_1 \cup L_2$ (simulate both TMs in parallel)
- **Intersection**: $L_1 \cap L_2$ (simulate both TMs sequentially)

- **Concatenation**: $L_1 \cdot L_2$
- **Kleene star**: L^*
- **Homomorphism**: h(L)
- **Inverse homomorphism**: h^{-1}(L)

RE Languages NOT closed under:

- **Complement**: $\overline{L}$ (if closed, all RE would be recursive)
- **Difference**: $L_1 \setminus L_2$

Recursive Languages closed under:

- All operations that RE is closed under
- **Complement**: $\overline{L}$ (since decision procedure exists)
- **Difference**: $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$

Undecidability Results:

Halting Problem (Turing, 1936):
Theorem: L_H = {(M, w) | TM M halts on input w} is undecidable

Proof (Diagonalization):
Assume H decides L_H. Construct TM D:

```
D(M):
  if H(M, M) accepts:  // M halts on itself
    loop forever
  else:
    halt
```

Consider D(D):

- If H(D, D) accepts: D loops forever, so D doesn't halt on D
- If H(D, D) rejects: D halts, so D halts on D
  Contradiction! H cannot exist.

Rice's Theorem:
Theorem: Any non-trivial property of RE languages is undecidable

Non-trivial property:

- Some RE language has the property
- Some RE language doesn't have the property

Examples of undecidable properties:

- Is L(M) = ∅? (Emptiness)

- Is L(M) finite? (Finiteness)
- Is L(M) regular? (Regularity)
- Is $L(M_1) = L(M_2)$? (Equivalence)
- Is $L(M_1) \subseteq L(M_2)$? (Containment)

Decidable properties (trivial):

- Is L(M) an RE language? (Yes, always)
- Is L(M) = L(M)? (Yes, always)

Reduction Techniques:

Many-one Reduction: $L_1 \leq_m L_2$
Exists computable function f such that:
$w \in L_1 \iff f(w) \in L_2$

Properties:

- If $L_1 \leq_m L_2$ and L_2 decidable, then L_1 decidable
- If $L_1 \leq_m L_2$ and L_1 undecidable, then L_2 undecidable
- Transitive: $L_1 \leq_m L_2 \leq_m L_3 \Rightarrow L_1 \leq_m L_3$

Common Reductions:

- Halting Problem ≤_m Emptiness Problem
- Halting Problem ≤_m Equivalence Problem
- Acceptance Problem ≤_m Halting Problem

Enumeration:

Theorem: L is RE ⟺ L can be enumerated by TM

Enumeration: TM that outputs strings of L (possibly with repetitions)

Proof (⇒): If L is RE, enumerate by dovetailing:

- Run TM on all strings in parallel
- Output string when TM accepts it

Proof (⇐): If L enumerated by TM E:

- To test $w \in L$: run E until w appears in enumeration

Hierarchy Results:

Arithmetic Hierarchy:

- $\Sigma_0 = \Pi_0$ = Recursive languages

- Σ_1 = RE languages
- Π_1 = co-RE languages
- Higher levels: more complex quantifier alternations

Post's Theorem: $\Sigma n \cup \Pi_n \subset \Sigma\{n+1\} \cap \Pi_{n+1}$

Degrees of Unsolvability:

- Turing degrees measure relative computability
- 0 = recursive languages
- 0' = Halting Problem degree
- Infinite hierarchy of degrees

Applications:

1. Compiler Theory:

- Syntax checking: decidable (CFL membership)
- Semantic analysis: often undecidable
- Optimization correctness: undecidable

2. Program Verification:

- Termination: undecidable (Halting Problem)
- Correctness: undecidable (Rice's Theorem)
- Model checking: decidable for finite systems

3. Logic and Mathematics:

- First-order logic: semi-decidable
- Arithmetic: undecidable (Gödel)
- Set theory: undecidable

4. Artificial Intelligence:

- Planning: undecidable in general
- Learning: PAC-learning has decidable cases
- Reasoning: depends on logic fragment

Practical Implications:

Semi-decision Procedures:

- May not terminate on negative instances
- Useful when positive answer expected
- Theorem provers, model checkers

Approximation Algorithms:

- When exact solution undecidable
- Heuristics and bounded search
- Satisficing vs optimizing

Bounded Versions:

- Time-bounded halting: decidable
- Space-bounded computation: decidable
- Resource-bounded complexity classes

GATE Tips:

- Recursive = decidable (always halts)
- RE = semi-decidable (may not halt on rejection)
- L recursive $\Leftrightarrow$ L and $\overline{L}$ both RE
- Halting Problem: RE but not recursive
- Rice's Theorem: non-trivial properties undecidable
- RE closed under $\cup$, $\cap$ but not complement
- Recursive closed under all Boolean operations
- Reduction: $L_1 \leq_m L_2$ transfers undecidability
- Enumeration characterizes RE languages
- Diagonalization proves undecidability

Problem-Solving Strategy:

To prove undecidability:

1. Reduce from known undecidable problem (usually Halting)
2. Use Rice's Theorem if property of RE languages
3. Diagonalization for fundamental problems

To prove decidability:

1. Construct explicit algorithm
2. Show algorithm always terminates
3. Prove correctness

Common Undecidable Problems:

- Halting Problem
- Emptiness of TM language
- Equivalence of TMs
- Totality (TM halts on all inputs)

- Regularity of TM language
- Context-freeness of TM language
- Finiteness of TM language
- Disjointness of TM languages

10.16 Reduction (2)

Key Concepts: Reductions are fundamental tools for comparing computational difficulty of problems. They transfer complexity and decidability properties between problems.

Many-One Reduction (Mapping Reduction):

Definition: Language A **many-one reduces** to language B (written $A \leq_m B$) if there exists a computable function $f: \Sigma \to \Sigma$ such that:
$\forall w \in \Sigma^*: w \in A \Leftrightarrow f(w) \in B$

Intuition:

- Transform instances of problem A into instances of problem B
- Preserve yes/no answers
- If we can solve B, we can solve A

Properties of Many-One Reduction:

1. **Reflexive**: $A \leq_m A$ (identity function)
2. **Transitive**: $A \leq_m B \wedge B \leq_m C \Rightarrow A \leq_m C$
3. **Decidability Transfer**: If $A \leq_m B$ and B decidable, then A decidable
4. **Undecidability Transfer**: If $A \leq_m B$ and A undecidable, then B undecidable
5. **RE Transfer**: If $A \leq_m B$ and B is RE, then A is RE

Turing Reduction (Oracle Reduction):

Definition: Language A **Turing reduces** to language B (written $A \leq_T B$) if there exists a Turing Machine M with oracle for B that decides A.

Oracle TM:

- Special "oracle tape" and states
- Can query oracle: "Is string x in B?"
- Oracle answers in one step
- More powerful than many-one reduction

Relationship: $A \leq_m B \Rightarrow A \leq_T B$ (but not vice versa)

Using Reductions to Prove Undecidability:

Method:

1. Choose known undecidable problem A (often Halting Problem)
2. Assume target problem B is decidable
3. Show A ≤_m B (construct reduction function f)
4. This would make A decidable (contradiction)
5. Therefore B is undecidable

Template:

```
To prove B undecidable:
1. Assume B is decidable with TM M_B
2. Construct reduction f: A → B where A is undecidable
3. Build TM M_A:
   M_A(w):
     compute f(w)
     return M_B(f(w))
4. M_A decides A, contradicting undecidability of A
5. Therefore B is undecidable
```

Classic Reduction Examples:

Example 1: Halting Problem ≤_m Emptiness Problem

Halting Problem: H = {(M, w) | TM M halts on input w}
Emptiness Problem: E = {M | L(M) = ∅}

Reduction: Given (M, w), construct M':

```
M'(x):  // for any input x
  simulate M on w
  if M halts on w:
    accept x
  // if M doesn't halt on w, M' loops forever
```

Analysis:

- If (M, w) ∈ H: M halts on w, so M' accepts all inputs, L(M') = Σ*, M' ∉ E
- If (M, w) ∉ H: M doesn't halt on w, so M' never accepts, L(M') = ∅, M' ∈ E

Therefore: (M, w) ∈ H ⟺ M' ∉ E ⟺ M' ∈ ¬E

So H ≤_m ¬E, which means ¬H ≤_m E.
Since ¬H is not RE (hence undecidable), E is undecidable.

Example 2: Acceptance Problem ≤_m Halting Problem

Acceptance Problem: A = {(M, w) | TM M accepts input w}
Halting Problem: H = {(M, w) | TM M halts on input w}

Reduction: Given (M, w), construct M':

```
M'(x):  // M' simulates M on w regardless of input x
  simulate M on w
  if M accepts w:
    accept x
  if M rejects w:
    reject x
  // if M loops on w, M' loops on x
```

Analysis:

- If (M, w) ∈ A: M accepts w, so M' halts on any input, (M', x) ∈ H for any x
- If (M, w) ∉ A: Either M rejects w (M' halts) or M loops on w (M' loops)

Refinement needed: Construct M' that halts iff M accepts w:

```
M'(x):
  simulate M on w
  if M accepts w:
    halt (accept)
  if M rejects w:
    loop forever
```

Now: (M, w) ∈ A ⟺ (M', x) ∈ H for any x

Rice's Theorem via Reduction:

Rice's Theorem: Any non-trivial property of RE languages is undecidable.

Proof Outline:

1. Let P be non-trivial property of RE languages
2. WLOG assume ∅ ∉ P (empty language doesn't have property)
3. Since P non-trivial, ∃ language L ∈ P
4. Reduce Halting Problem to P:
   - Given (M, w), construct M_w that accepts L if M halts on w, else accepts ∅
   - (M, w) ∈ H ⟺ L(M_w) ∈ P
5. This shows H ≤_m P, so P undecidable

Reduction Construction Techniques:

1. Simulation Technique:

- New TM simulates original TM
- Modify behavior based on simulation result

- Used in Halting → Emptiness reduction

2. Encoding Technique:

- Encode problem instance in TM description
- TM behavior depends on encoded information
- Used in many Rice's Theorem applications

3. Gadget Construction:

- Build "gadgets" that enforce constraints
- Combine gadgets to simulate original problem
- Common in NP-completeness proofs

4. Local Replacement:

- Replace parts of instance with equivalent parts
- Preserve overall problem structure
- Used in graph problems

Complexity-Theoretic Reductions:

Polynomial-Time Reduction: $A \leq_p B$

- Reduction function computable in polynomial time
- Used for NP-completeness
- Preserves polynomial-time solvability

Log-Space Reduction: $A \leq_L B$

- Reduction function computable in logarithmic space
- Stronger than polynomial-time reduction
- Used for space complexity classes

AC^0 Reduction:

- Reduction computable by constant-depth circuits
- Very restrictive but useful for fine-grained complexity

Applications of Reductions:

1. Proving Undecidability:

- Systematic way to show problems undecidable
- Build undecidability hierarchy
- Rice's Theorem applications

2. Complexity Classification:

- Show problems are NP-complete
- Establish complexity hierarchies
- Fine-grained complexity analysis

3. Algorithm Design:

- Reduce new problem to known solvable problem
- Transform algorithms between domains
- Approximation algorithm design

4. Lower Bounds:

- Show problems require certain resources
- Conditional lower bounds
- Hardness of approximation

Common Mistakes in Reductions:

1. Wrong Direction:

- To prove B hard, need A ≤ B where A is hard
- Not B ≤ A

2. Non-Computable Reduction:

- Reduction function must be computable
- Often need effective construction

3. Not Preserving Structure:

- Must preserve yes/no answers
- Check both directions of equivalence

4. Circular Reasoning:

- Can't use undecidability of B to prove B undecidable
- Need independent undecidable problem

GATE Tips:

- Reduction $A \leq_m B$: solving B helps solve A
- To prove B undecidable: reduce known undecidable A to B
- Halting Problem is common source for reductions
- Rice's Theorem: non-trivial properties of RE languages undecidable
- Reduction must be computable function
- Preserve yes/no answers: $w \in A \Leftrightarrow f(w) \in B$
- Transitivity: $A \leq B \leq C$ implies $A \leq C$

- Many-one reduction stronger than Turing reduction
- Construction often involves building new TM

Problem-Solving Strategy:

To construct reduction A ≤_m B:

1. Understand both problems A and B clearly
2. Given instance w of A, construct f(w) for B
3. Ensure $w \in A \Leftrightarrow f(w) \in B$
4. Verify f is computable
5. Check both directions of equivalence

To prove undecidability:

1. Choose appropriate undecidable problem A
2. Construct reduction A ≤_m B
3. Conclude B undecidable
4. Often use Halting Problem or Rice's Theorem

Standard Undecidable Problems for Reductions:

- Halting Problem: {(M, w) | M halts on w}
- Acceptance Problem: {(M, w) | M accepts w}
- Emptiness: {M | L(M) = ∅}
- Totality: {M | L(M) = Σ*}
- Equivalence: {(M_1, M_2) | $L(M_1) = L(M_2)$}
- Regularity: {M | L(M) is regular}
- Finiteness: {M | L(M) is finite}

10.17 Regular Expression (29)

Key Concepts: Regular expressions provide a declarative way to specify regular languages using algebraic notation. They are equivalent in power to finite automata but more intuitive for pattern specification.

Regular Expression Definition:

Base Cases:

- ∅ (empty set): Denotes empty language {}
- **ε** (epsilon): Denotes language {ε} containing only empty string
- **a** (symbol): For a ∈ Σ, denotes language {a}

Inductive Cases:

- **Union (R|S)**: L(R|S) = L(R) ∪ L(S)
- **Concatenation (RS)**: L(RS) = L(R) · L(S) = {xy | x ∈ L(R), y ∈ L(S)}
- **Kleene Star (R*)**: L(R*) = *(L(R))* = ∪_{i=0}^{∞} (L(R))^i

Precedence (highest to lowest):

1. Kleene star (*)
2. Concatenation (implicit)
3. Union (|)

Parentheses override precedence: (R|S)*T* = *((R|S))*T

Extended Regular Expressions:

Additional Operators (for convenience):

- **R+**: One or more = RR = *R*R
- **R?**: Zero or one = R|ε
- **R{n}**: Exactly n repetitions
- **R{m,n}**: Between m and n repetitions
- **[abc]**: Character class = a|b|c
- **[a-z]**: Character range
- **abc**: Negated character class
- **.**: Any character (wildcard)

Examples:

Example 1: (a|b)*abb

- Strings over {a,b} ending in "abb"
- L = {abb, aabb, babb, ababb, baabb, ...}

Example 2: a*b*

- Zero or more a's followed by zero or more b's
- L = {ε, a, b, aa, ab, bb, aaa, aab, abb, bbb, ...}

Example 3: (a|b)*a(a|b)*

- Strings containing at least one 'a'
- Complement of b* (strings with no 'a')

Example 4: ((a|b)(a|b))*

- Strings of even length
- Each pair can be aa, ab, ba, or bb

Regular Expression to Automaton:

Thompson's Construction (Regex to ε-NFA):

Base Cases:

- ∅: Two states, no transitions, no final states
- **ε**: Two states, ε-transition between them, second is final
- **a**: Two states, transition labeled 'a', second is final

Inductive Cases:

- **Union R|S**:
  - New start state with ε-transitions to starts of R and S
  - New final state with ε-transitions from finals of R and S
- **Concatenation RS**:
  - Connect final states of R to start state of S with ε-transitions
  - Start of R becomes start, final of S becomes final
- **Kleene Star R***:
  - New start state (also final) with ε-transition to start of R
  - ε-transitions from finals of R back to start of R
  - ε-transition from new start to new final (for ε)

Properties:

- Resulting ε-NFA has O(|r|) states for regex r
- Each state has at most 2 outgoing ε-transitions
- Construction is compositional and modular

Automaton to Regular Expression:

State Elimination Method:

1. Add new start state with ε-transition to old start
2. Add new final state with ε-transitions from old finals
3. Eliminate intermediate states one by one:
   - For eliminated state q with incoming edges labeled R_i and outgoing edges labeled S_j
   - Add direct edges labeled R_i(loop at q)*S_j
4. Final regex is label on edge from new start to new final

Arden's Theorem: If $X = AX \cup B$ and $\varepsilon \notin L(A)$, then $X = A^*B$

Algebraic Laws for Regular Expressions:

Associativity:

- (R|S)|T = R|(S|T)
- (RS)T = R(ST)

Commutativity:

- R|S = S|R
- RS ≠ SR (concatenation not commutative)

Identity Elements:

- R|∅ = R
- Rε = εR = R
- R∅ = ∅R = ∅

Idempotence:

- R|R = R
- RR ≠ R (concatenation not idempotent)

Distributivity:

- R(S|T) = RS|RT
- (S|T)R = SR|TR
- R|(S&T) ≠ (R|S)&(R|T) (union doesn't distribute over intersection)

Kleene Star Laws:

- (R*)* = R*
- ∅* = ε
- ε* = ε
- R* = ε|*RR**
- R* = ε|*R*R*
- (R|S)* = *(R*S)**

De Morgan's Laws (don't apply directly to regex):

- Regular expressions don't have complement operator
- Need to work with languages: L(R|S) = L(R) ∪ L(S)

Practical Regular Expression Features:

Anchors:

- **^**: Start of string
- **$**: End of string
- **\b**: Word boundary

Quantifiers:

- **{n}**: Exactly n times
- **{n,}**: At least n times
- **{n,m}**: Between n and m times
- **?**: 0 or 1 (non-greedy: ??)
- **+**: 1 or more (non-greedy: +?)
- *: 0 or more (non-greedy: *?)

Character Classes:

- **\d**: Digits [0-9]
- **\w**: Word characters [a-zA-Z0-9_]
- **\s**: Whitespace [\t\n\r\f]
- **\D, \W, \S**: Negated versions

Grouping and Capturing:

- **(pattern)**: Capturing group
- **(?:pattern)**: Non-capturing group
- **\1, \2**: Back-references to captured groups

Lookahead/Lookbehind:

- **(?=pattern)**: Positive lookahead
- **(?!pattern)**: Negative lookahead
- **(?<=pattern)**: Positive lookbehind
- **(?<!pattern)**: Negative lookbehind

Applications:

1. Text Processing:

- Search and replace operations
- Data validation (email, phone numbers)
- Log file analysis
- Pattern extraction

2. Lexical Analysis:

- Token recognition in compilers
- Keyword identification
- Number and string literal parsing
- Comment removal

3. Bioinformatics:

- DNA sequence pattern matching
- Protein motif identification
- Restriction enzyme site finding
- Phylogenetic analysis

4. Network Security:

- Intrusion detection patterns
- Malware signature matching
- URL filtering
- Protocol analysis

5. Data Mining:

- Information extraction
- Web scraping
- Document classification
- Feature extraction

Regular Expression Engines:

NFA-based Engines:

- Implement Thompson construction
- Guaranteed linear time O(nm)
- Support all regex features
- Examples: grep, awk

DFA-based Engines:

- Convert to DFA for faster matching
- May have exponential space
- Very fast for simple patterns
- Examples: lex, flex

Backtracking Engines:

- Recursive descent with backtracking
- Support advanced features (backreferences)
- Can have exponential time complexity
- Examples: Perl, Python, Java

Hybrid Engines:

- Combine multiple approaches
- Optimize for common cases

- Examples: RE2, Rust regex

Performance Considerations:

Catastrophic Backtracking:

- Patterns like (a+)+b on input "aaaa...c"
- Exponential time due to backtracking
- Avoid nested quantifiers

Optimization Techniques:

- **Anchoring**: Use ^ and $ when possible
- **Specificity**: More specific patterns first
- **Character classes**: [abc] faster than (a|b|c)
- **Non-capturing groups**: (?:...) when capture not needed
- **Atomic groups**: (?>...) prevent backtracking

Common Patterns:

Email Validation (simplified):

```
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
```

IP Address:

```
((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9]
[0-9]?)
```

Phone Number (US format):

```
\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})
```

URL:

```
https?://[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}(/[^\s]*)?
```

GATE Tips:

- Regular expressions = Regular languages = Finite automata
- Thompson construction: regex to ε-NFA in O(|r|) states
- State elimination: automaton to regex
- Precedence: * > concatenation > |
- Kleene star: R = $\varepsilon | RR$ = ε|R*R

- Union: R|S (either R or S)
- Concatenation: RS (R followed by S)
- Empty set ∅ vs empty string ε
- (R*) = R*
- Distributivity: R(S|T) = RS|RT
- Regular expressions closed under union, concatenation, star

Common GATE Patterns:

Strings ending in pattern:

- Ending in "01": (a|b)*01
- Ending in "abb": (a|b)*abb

Strings containing pattern:

- Containing "01": (a|b)*01(a|b)*
- Containing substring: Σ*pattern*Σ*

Counting patterns:

- Even number of a's: b*(ab*ab)*
- Odd number of a's: b*a(b*ab*ab*)*

Length constraints:

- Strings of length 3: (a|b)(a|b)(a|b)
- Strings of even length: ((a|b)(a|b))*

Complement patterns:

- Not containing "aa": ε|a|(ba)*(b|ε)
- Not ending in "01": Use DFA construction then convert back

Debugging Regular Expressions:

- Test with simple examples
- Check edge cases (ε, single characters)
- Verify precedence with parentheses
- Use visualization tools
- Break complex patterns into parts

10.18 Regular Grammar (3)

Key Concepts: Right-linear A→wB or w.

10.19 Regular Language (36)

Key Concepts: Languages accepted by DFA/NFA/regex. Closed under union, intersection, complement, concatenation, Kleene star.

Pumping Lemma for Regular Languages:

For every regular language L, ∃ pumping length p such that:

- Every string $w \in L$ with $|w| \geq p$ can be divided as $w = xyz$ where:
  1. $|xy| \leq p$ (pump early)
  2. $|y| \geq 1$ (pump something)
  3. $\forall k \geq 0 : xy^k z \in L$ (pumping preserves membership)

Using Pumping Lemma to Prove Non-Regularity:

Proof by contradiction:

1. Assume L is regular
2. Let p be the pumping length
3. Choose string $w \in L$ with $|w| \geq p$ (strategic choice!)
4. For all possible divisions $w = xyz$ satisfying conditions 1 & 2
5. Show ∃k where $xy^k z \notin L$ (contradiction)
6. Therefore L is not regular

Strategy for Choosing w:

- Use p in the string definition
- Common patterns: $a^p b^p$, a^{p^2}, a^p!
- Make pumping force violation

Classic Non-Regular Languages:

1. $L = \{a^n b^n | n \geq 0\}$
   - Choose $w = a^p b^p$
   - y must be in a's (since $|xy| leq p$)
   - Pump: $xy^2 z$ has more a's than b's ✗
2. $L = \{ww | w \in \{a, b\}^*\}$ (palindromes)
   - Choose $w = a^p b a^p$
   - Pumping y (all a's) breaks symmetry
3. $L = \{a^{n^2} | n \geq 0\}$ (perfect squares)
   - Choose $w = a^{p^2}$
   - After pumping: $p^2 < |xy^2 z| < (p+1)^2$ ✗
4. $L = \{a^p | p \text{ is prime}\}$
   - Choose $w = a^q$ where $q \geq p$ is prime
   - Pump k=q times: $|xy^q z| = q + (q-1)|y| = q|y|$ (composite) ✗

Closure Properties:

Regular languages are closed under:

- **Union**: $L_1 \cup L_2$ (DFA: product construction with OR)
- **Intersection**: $L_1 \cap L_2$ (DFA: product construction with AND)
- **Complement**: $\overline{L}$ (DFA: flip accept states)
- **Concatenation**: $L_1 \cdot L_2$ (NFA: ε from L_1 accept to L_2 start)
- **Kleene Star**: L^* (NFA: ε loops)
- **Reversal**: L^R (Reverse all transitions)
- **Difference**: $L_1 - L_2 = L_1 \cap \overline{L_2}$
- **Homomorphism**: $h(L)$ where $h : \Sigma^* \to \Gamma^*$
- **Inverse Homomorphism**: $h^{-1}(L)$

NOT closed under:

- None! (Regular languages are closed under all standard operations)

Myhill-Nerode Theorem:

L is regular ⟺ The number of equivalence classes of $\equiv_L$ is finite

- $x \equiv_L y$ iff $\forall z : xz \in L \Leftrightarrow yz \in L$
- Number of classes = minimum number of states in DFA
- Use to prove non-regularity or find minimum DFA

Decision Problems (all decidable for regular languages):

- **Membership**: Is $winL$? (Simulate DFA)
- **Emptiness**: Is $L = emptyset$? (Check if accept state reachable)
- **Finiteness**: Is $|L| < infty$? (Check for cycles to accept states)
- **Equivalence**: Is $L_1 = L_2$? (Check if $(L_1 - L_2)cup(L_2 - L_1) = emptyset$)
- **Containment**: Is $L_1 subseteq L_2$? (Check if $L_1 - L_2 = emptyset$)

Distinguishing Strings:

- String z distinguishes x and y if exactly one of xz, yz is in L
- If k strings pairwise distinguishable → DFA needs ≥k states

Regular Language Examples:

- $\{w | w \text{ has even number of } a\text{'s}\}$ ✓
- $\{a^{2n} | n \geq 0\}$ ✓
- $(a|b)^*abb$ ✓ (strings ending in abb)
- $\{w | \#_a(w) = \#_b(w)\}$ ✗ (counting requires memory)
- $\{w | w = w^R\}$ (palindromes) ✗ (needs stack)

GATE Tips:

- To prove non-regularity: Use pumping lemma or Myhill-Nerode
- Pumping lemma is only for proving non-regularity (not sufficiency)
- Choose w strategically - make y's position matter
- Remember: $|xy| \leq p$ restricts where y can be
- Closure properties useful for constructing/disproving regularity
- Minimum states = number of Myhill-Nerode classes

Common Mistakes:

- Trying to prove regularity with pumping lemma (can't!)
- Forgetting $|xy| \leq p$ constraint
- Not considering all possible divisions of w
- Showing one division fails (need to show ALL fail)

Missing Topics - Computer Networks

DHCP (Dynamic Host Configuration Protocol)

Key Concepts: DHCP automates IP address assignment and network configuration for hosts joining a network. Essential for scalable network management.

DHCP Overview:

- **Purpose**: Automatically assign IP addresses and network parameters to hosts
- **Port**: UDP 67 (server), UDP 68 (client)
- **Application Layer Protocol**: Works over UDP/IP
- **Eliminates**: Manual IP configuration, reduces configuration errors

DHCP Components:

1. DHCP Server:

- Maintains pool of available IP addresses
- Stores configuration parameters (subnet mask, gateway, DNS servers)
- Manages lease duration and renewal

2. DHCP Client:

- Host requesting network configuration
- Sends DHCP messages to obtain IP address

3. DHCP Relay Agent:

- Forwards DHCP messages between clients and servers on different subnets

- Necessary because DHCP uses broadcast (doesn't cross routers)

DHCP Message Types:

1. **DHCPDISCOVER**: Client broadcasts to find DHCP servers
2. **DHCPOFFER**: Server offers IP address to client
3. **DHCPREQUEST**: Client requests offered IP address
4. **DHCPACK**: Server acknowledges and confirms assignment
5. **DHCPNAK**: Server denies request (address no longer available)
6. **DHCPRELEASE**: Client releases IP address
7. **DHCPDECLINE**: Client declines offered address (address already in use)
8. **DHCPINFORM**: Client requests additional configuration (already has IP)

DHCP Process (DORA):

4-Step Process:

1. **Discover**: Client broadcasts DHCPDISCOVER
   - Source: 0.0.0.0:68
   - Destination: 255.255.255.255:67
   - Message: "I need an IP address"
2. **Offer**: Server responds with DHCPOFFER
   - Contains: Available IP address, lease time, subnet mask, gateway, DNS
   - May receive multiple offers from different servers
3. **Request**: Client broadcasts DHCPREQUEST
   - Accepts one offer, implicitly rejects others
   - Broadcasts so other servers know their offers were declined
4. **Acknowledge**: Server sends DHCPACK
   - Confirms IP address assignment
   - Client can now use the IP address

DHCP Lease:

Lease Duration: Time period for which IP address is assigned

- Typical: Hours to days
- **Lease Renewal**: Client attempts to renew at 50% of lease time (T1)
- **Rebinding**: If renewal fails, tries at 87.5% of lease time (T2)
- **Expiration**: If no renewal, client must release IP and restart DORA

DHCP Configuration Parameters:

Provided to client:

- **IP Address**: Assigned to client

- **Subnet Mask**: Network/host boundary
- **Default Gateway**: Router IP for external communication
- **DNS Servers**: Domain name resolution
- **Lease Time**: Duration of assignment
- **Domain Name**: Local domain suffix
- **NTP Servers**: Time synchronization (optional)

DHCP Address Allocation:

1. Dynamic Allocation:

- IP assigned from pool for limited time
- Most common method
- Address returned to pool after lease expires

2. Automatic Allocation:

- IP assigned permanently from pool
- Same IP given to client each time (based on MAC)

3. Static Allocation:

- Manual binding of IP to MAC address
- Administrator configures specific IP for specific device
- Used for servers, printers, network devices

DHCP Relay:

Problem: DHCP uses broadcast, which doesn't cross routers

Solution: DHCP Relay Agent

- Router interface configured as relay
- Receives broadcast DHCPDISCOVER
- Converts to unicast and forwards to DHCP server
- Server responds to relay, which forwards to client

DHCP Security Issues:

1. DHCP Spoofing:

- Rogue DHCP server provides false configuration
- Can redirect traffic through attacker

2. DHCP Starvation:

- Attacker requests all available IPs

- Legitimate clients cannot obtain addresses

3. Mitigation:

- **DHCP Snooping**: Switch feature that validates DHCP messages
- **Port Security**: Limit DHCP servers to trusted ports
- **Rate Limiting**: Limit DHCP requests per port

DHCP vs Static IP:

Feature	DHCP	Static IP
Configuration	Automatic	Manual
Scalability	High	Low
Management	Centralized	Distributed
Flexibility	High	Low
Use Case	End users	Servers, network devices

GATE Tips:

- DHCP uses UDP ports 67 (server) and 68 (client)
- DORA process: Discover, Offer, Request, Acknowledge
- Client broadcasts DISCOVER and REQUEST
- Lease renewal at 50% (T1), rebinding at 87.5% (T2)
- DHCP relay needed for clients on different subnet than server
- Dynamic allocation most common, static for servers
- Security: DHCP snooping prevents rogue servers

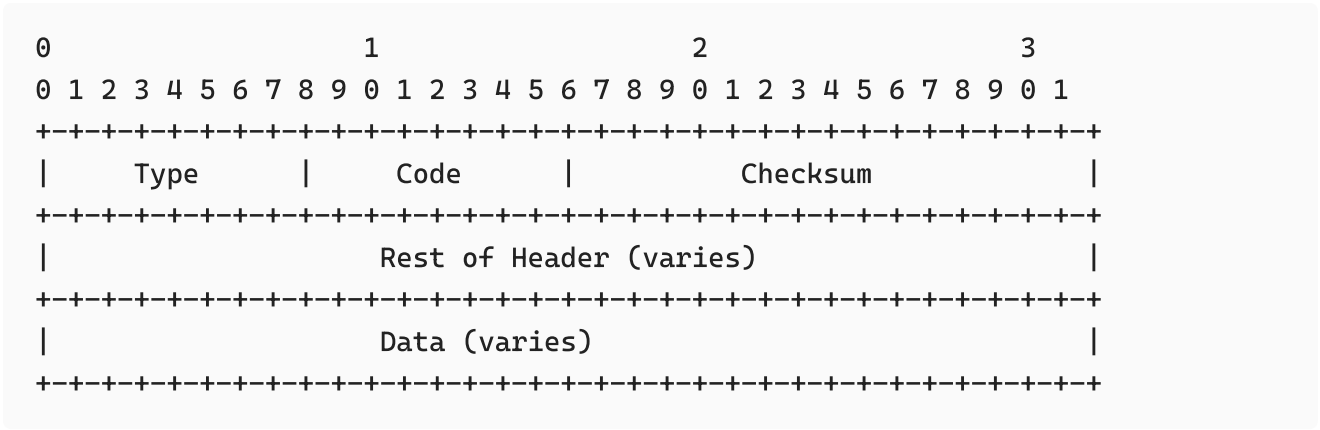
ICMP (Internet Control Message Protocol)

Key Concepts: ICMP provides error reporting and diagnostic functions for IP. Essential for network troubleshooting and management.

ICMP Overview:

- **Layer**: Network Layer (Layer 3)
- **Protocol Number**: 1 (in IP header)
- **Purpose**: Error reporting, diagnostics, network testing
- **Encapsulation**: ICMP message encapsulated in IP datagram
- **Not for**: Data transfer (control protocol only)

ICMP Message Format:



Fields:

- **Type**: ICMP message type (8 bits)
- **Code**: Subtype within message type (8 bits)
- **Checksum**: Error detection for ICMP message (16 bits)
- **Rest of Header**: Varies by message type
- **Data**: Original IP header + first 8 bytes of data (for error messages)

ICMP Message Types:

Error Reporting Messages:

1. Destination Unreachable (Type 3):

- **Code 0**: Network unreachable
- **Code 1**: Host unreachable
- **Code 2**: Protocol unreachable
- **Code 3**: Port unreachable
- **Code 4**: Fragmentation needed but DF set
- **Code 5**: Source route failed
- **Code 6**: Destination network unknown
- **Code 7**: Destination host unknown

2. Time Exceeded (Type 11):

- **Code 0**: TTL expired in transit (used by traceroute)
- **Code 1**: Fragment reassembly time exceeded

3. Parameter Problem (Type 12):

- **Code 0**: Pointer indicates error in IP header
- **Code 1**: Missing required option

4. Source Quench (Type 4) - Deprecated:

- Congestion control (no longer used)

5. Redirect (Type 5):

- **Code 0**: Redirect for network
- **Code 1**: Redirect for host
- Router informs host of better route

Query Messages:

1. Echo Request (Type 8) / Echo Reply (Type 0):

- **Ping utility**: Tests reachability
- Request sent with identifier and sequence number
- Reply echoes back same data
- **Format**:
  - Identifier: Process ID
  - Sequence Number: Increments with each ping
  - Data: Optional payload

2. Timestamp Request (Type 13) / Timestamp Reply (Type 14):

- Measure round-trip time
- Synchronize clocks
- Contains: Originate, Receive, Transmit timestamps

3. Address Mask Request (Type 17) / Address Mask Reply (Type 18):

- Host queries router for subnet mask
- Rarely used (DHCP provides this)

4. Router Solicitation (Type 10) / Router Advertisement (Type 9):

- Host discovers routers on network
- Router announces its presence

ICMP Applications:

1. Ping:

- Tests host reachability
- Measures round-trip time (RTT)
- Uses Echo Request/Reply (Type 8/0)
- **Process**:
  1. Send Echo Request to destination
  2. Destination responds with Echo Reply
  3. Calculate RTT from send/receive times
- **Output**: Sequence number, TTL, RTT

2. Traceroute (tracert on Windows):

- Discovers path to destination
- Uses TTL expiration and ICMP Time Exceeded
- **Process**:
  1. Send packet with TTL=1
  2. First router decrements TTL to 0, sends Time Exceeded
  3. Send packet with TTL=2
  4. Second router sends Time Exceeded
  5. Continue until destination reached
- **Output**: List of routers (hops) to destination

3. Path MTU Discovery:

- Determines maximum transmission unit along path
- Uses Destination Unreachable (Fragmentation Needed)
- **Process**:
  1. Send packet with DF (Don't Fragment) flag set
  2. If too large, router sends ICMP Fragmentation Needed
  3. Reduce packet size and retry
  4. Converge on maximum MTU

ICMP Error Message Rules:

ICMP errors NOT generated for:

1. ICMP error messages (prevents infinite loops)
2. Fragmented datagrams (except first fragment)
3. Multicast/broadcast datagrams
4. Link-layer broadcast frames
5. Datagrams with source address 0.0.0.0 or loopback

Why: Prevent ICMP storms and network congestion

ICMP and Security:

Security Concerns:

1. ICMP Flood (Ping Flood):

- Overwhelm target with Echo Requests
- DDoS attack vector

2. Ping of Death:

- Send oversized ICMP packet

- Causes buffer overflow (historical vulnerability)

3. Smurf Attack:

- Send Echo Request to broadcast address with spoofed source
- All hosts reply to victim
- Amplification attack

4. ICMP Redirect Attack:

- Malicious router sends false redirects
- Hijacks traffic

5. ICMP Tunneling:

- Covert channel for data exfiltration
- Embed data in ICMP packets

Mitigation:

- **Rate limiting**: Limit ICMP traffic
- **Filtering**: Block unnecessary ICMP types
- **Disable broadcast**: Prevent Smurf attacks
- **Ingress filtering**: Validate source addresses

ICMPv6:

Differences from ICMPv4:

- **Protocol Number**: 58 (instead of 1)
- **Mandatory**: Required for IPv6 (not optional)
- **Additional Functions**:
  - Neighbor Discovery (replaces ARP)
  - Path MTU Discovery (mandatory)
  - Multicast Listener Discovery

New Message Types:

- Neighbor Solicitation (Type 135)
- Neighbor Advertisement (Type 136)
- Router Solicitation (Type 133)
- Router Advertisement (Type 134)

GATE Tips:

- ICMP is Network Layer protocol (IP protocol number 1)
- Ping uses Echo Request (Type 8) and Echo Reply (Type 0)

- Traceroute uses TTL expiration and Time Exceeded (Type 11)
- Destination Unreachable (Type 3) has multiple codes
- ICMP errors include original IP header + 8 bytes of data
- No ICMP error for ICMP error (prevents loops)
- Path MTU Discovery uses Fragmentation Needed (Type 3, Code 4)
- Security: ICMP can be used for attacks (flood, Smurf, tunneling)

NAT (Network Address Translation)

Key Concepts: NAT translates private IP addresses to public IP addresses, enabling multiple devices to share a single public IP. Critical for IPv4 address conservation.

NAT Overview:

- **Purpose**: Map private IPs to public IPs
- **Location**: Typically on router/gateway
- **Motivation**: IPv4 address exhaustion
- **Benefit**: Security (hides internal network structure)

Private IP Address Ranges (RFC 1918):

Non-routable on public Internet:

- **Class A**: 10.0.0.0 to 10.255.255.255 (10.0.0.0/8)
- **Class B**: 172.16.0.0 to 172.31.255.255 (172.16.0.0/12)
- **Class C**: 192.168.0.0 to 192.168.255.255 (192.168.0.0/16)

NAT Types:

1. Static NAT (One-to-One):

- **Mapping**: One private IP ↔ One public IP
- **Permanent**: Mapping configured manually
- **Use Case**: Servers that need consistent public IP
- **Example**:
  - Internal: 192.168.1.10 → External: 203.0.113.5
  - Always same mapping

2. Dynamic NAT (One-to-One Pool):

- **Mapping**: Private IPs mapped to pool of public IPs
- **Temporary**: Mapping created on-demand
- **Limitation**: Number of simultaneous connections ≤ pool size
- **Example**:
  - Pool: 203.0.113.5 - 203.0.113.10 (6 addresses)

- First 6 internal hosts get public IPs
- 7th host must wait

3. PAT (Port Address Translation) / NAT Overload:

- **Mapping**: Many private IPs ↔ One public IP (using ports)
- **Most Common**: Used in home/office routers
- **Mechanism**: Translate (private IP, private port) to (public IP, public port)
- **Capacity**: ~65,000 simultaneous connections per public IP

NAT Translation Table:

For PAT/NAT Overload:

Inside Local	Inside Local Port	Inside Global	Inside Global Port	Outside Global	Outside Global Port
192.168.1.10	3000	203.0.113.5	50000	93.184.216.34	80
192.168.1.11	3001	203.0.113.5	50001	93.184.216.34	80
192.168.1.10	3002	203.0.113.5	50002	151.101.1.69	443

Terminology:

- **Inside Local**: Private IP address
- **Inside Global**: Public IP address (after NAT)
- **Outside Global**: Destination public IP
- **Outside Local**: Destination as seen from inside (usually same as Outside Global)

NAT Operation:

Outbound (Inside to Outside):

1. Host 192.168.1.10:3000 sends packet to 93.184.216.34:80
2. NAT router receives packet
3. Router creates entry in NAT table
4. Router replaces source IP:port with public IP:port (203.0.113.5:50000)
5. Router forwards packet to Internet

Inbound (Outside to Inside):

1. Reply arrives at router: destination 203.0.113.5:50000
2. Router looks up port 50000 in NAT table
3. Router finds mapping to 192.168.1.10:3000
4. Router replaces destination IP:port with private IP:port

5. Router forwards packet to internal host

NAT Advantages:

1. **Address Conservation**: Multiple devices share one public IP
2. **Security**: Internal network structure hidden
3. **Flexibility**: Change ISP without renumbering internal network
4. **Cost**: Fewer public IPs needed

NAT Disadvantages:

1. **Breaks End-to-End Connectivity**: Violates Internet architecture principle
2. **Port Exhaustion**: Limited to ~65,000 simultaneous connections per public IP
3. **Protocol Issues**: Some protocols embed IP addresses in payload (FTP, SIP)
4. **Performance**: Translation overhead
5. **Complicates Peer-to-Peer**: Incoming connections difficult
6. **Logging**: Multiple users share same public IP

NAT Traversal Techniques:

Problem: Incoming connections to NATed hosts

Solutions:

1. Port Forwarding (Static Port Mapping):

- Manually configure router to forward specific port to internal host
- Example: Forward port 80 to 192.168.1.10:80 for web server

2. UPnP (Universal Plug and Play):

- Automatic port forwarding
- Application requests router to open port
- Security risk if not controlled

3. STUN (Session Traversal Utilities for NAT):

- Discover public IP and port
- Used for VoIP, video conferencing
- Works for cone NAT, not symmetric NAT

4. TURN (Traversal Using Relays around NAT):

- Relay server forwards traffic
- Works for all NAT types
- Higher latency and cost

5. ICE (Interactive Connectivity Establishment):

- Combines STUN and TURN
- Tries direct connection first, falls back to relay

NAT Types (Behavior):

1. Full Cone NAT:

- Once internal address mapped, any external host can send to that port
- Most permissive

2. Restricted Cone NAT:

- External host can send only if internal host sent to that external IP first
- Port doesn't matter

3. Port-Restricted Cone NAT:

- External host can send only if internal host sent to that exact IP:port
- Most common in home routers

4. Symmetric NAT:

- Different mapping for each destination
- Most restrictive, hardest for NAT traversal

NAT and IPv6:

IPv6 Eliminates Need for NAT:

- Abundant addresses (2^128)
- Every device can have public IP
- Restores end-to-end connectivity

NAT66 (IPv6-to-IPv6 NAT):

- Sometimes used for privacy or multihoming
- Controversial (defeats IPv6 purpose)

GATE Tips:

- NAT translates private IPs to public IPs
- Private ranges: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16
- PAT (NAT Overload) uses ports to multiplex connections
- Static NAT: one-to-one permanent mapping
- Dynamic NAT: one-to-one from pool
- PAT: many-to-one using ports

- NAT table maps (private IP, port) to (public IP, port)
- Advantages: address conservation, security
- Disadvantages: breaks end-to-end, complicates P2P
- Port forwarding enables incoming connections
- IPv6 eliminates need for NAT

Link State Routing

Key Concepts: Link state routing protocols build complete network topology map at each router, enabling optimal path calculation. OSPF is the primary example.

Link State Overview:

- **Philosophy**: Each router knows complete network topology
- **Method**: Flood link state information, build topology database
- **Algorithm**: Dijkstra's shortest path algorithm
- **Examples**: OSPF (Open Shortest Path First), IS-IS

Comparison with Distance Vector:

Feature	Distance Vector	Link State
Information	Distance to destinations	Complete topology
Updates	Periodic, triggered	Event-driven
Convergence	Slow (count-to-infinity)	Fast
Memory	Low	High
CPU	Low	High
Scalability	Limited	Better
Loop Prevention	Split horizon, poison reverse	Topology knowledge

Link State Algorithm Steps:

1. Discover Neighbors:

- Send Hello packets on all interfaces
- Learn directly connected neighbors
- Establish neighbor relationships

2. Measure Link Costs:

- Determine cost to each neighbor
- Based on bandwidth, delay, or administrative weight
- **OSPF Cost**: 10^8 / bandwidth (bps)

3. Build Link State Packet (LSP):

- Contains router ID and list of neighbors with costs
- Sequence number for freshness
- Age field for expiration

4. Flood LSPs:

- Send LSP to all neighbors
- Neighbors forward to their neighbors (except sender)
- Reliable flooding with acknowledgments
- Duplicate detection using sequence numbers

5. Build Topology Database:

- Collect LSPs from all routers
- Build complete network graph
- All routers have identical database (in steady state)

6. Run Dijkstra's Algorithm:

- Calculate shortest path tree rooted at self
- Determine next hop for each destination
- Build forwarding table

Link State Packet (LSP) Format:

```
+------------------+
| Router ID        |
+------------------+
| Sequence Number  |
+------------------+
| Age              |
+------------------+
| Neighbor 1 ID    |
| Cost to Neighbor 1|
+------------------+
| Neighbor 2 ID    |
| Cost to Neighbor 2|
+------------------+
| ...              |
+------------------+
```

Dijkstra's Algorithm for Link State:

```
Dijkstra(Graph G, Source s):
    Initialize:
```

```
        distance[s] = 0
        distance[v] = ∞ for all v ≠ s
        previous[v] = undefined for all v
        Q = set of all vertices

    while Q is not empty:
        u = vertex in Q with minimum distance[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt = distance[u] + weight(u, v)
            if alt < distance[v]:
                distance[v] = alt
                previous[v] = u

    return distance[], previous[]
```

Example Network:

```
    A ----5---- B
    |           |
    2           1
    |           |
    C ----3---- D
```

Router A's LSP:

- Router ID: A
- Neighbors: B (cost 5), C (cost 2)

Router B's LSP:

- Router ID: B
- Neighbors: A (cost 5), D (cost 1)

Dijkstra from A:

1. **Initial**: A(0), B(∞), C(∞), D(∞)
2. **Process A**: A(0), B(5), C(2), D(∞)
3. **Process C**: A(0), B(5), C(2), D(5)
4. **Process B**: A(0), B(5), C(2), D(5)
5. **Process D**: A(0), B(5), C(2), D(5)

Shortest Paths from A:

- A → B: A → B (cost 5) or A → C → D → B (cost 6) → Choose A → B
- A → C: A → C (cost 2)

- A → D: A → C → D (cost 5) or A → B → D (cost 6) → Choose A → C → D

LSP Flooding Process:

Reliable Flooding:

1. Router generates LSP with new sequence number
2. Sends LSP to all neighbors
3. Each neighbor:
   - Checks if LSP is newer (higher sequence number)
   - If newer, stores LSP and forwards to all neighbors except sender
   - Sends acknowledgment back to sender
4. If no acknowledgment received, retransmit LSP

Duplicate Detection:

- Each LSP has unique (Router ID, Sequence Number)
- Router maintains database of latest LSP from each router
- Ignore LSPs with older sequence numbers

LSP Aging:

- Each LSP has age field (time to live)
- Age decremented periodically
- When age reaches 0, LSP is purged from database
- Originating router must refresh LSP before expiration

Advantages of Link State:

1. **Fast Convergence**:
   - Event-driven updates
   - No count-to-infinity problem
   - Convergence time = flood time + computation time
2. **Loop-Free**:
   - Complete topology knowledge prevents loops
   - Consistent view of network
3. **Scalable**:
   - Hierarchical design possible (OSPF areas)
   - Efficient use of bandwidth (only changes flooded)
4. **Multiple Metrics**:
   - Can optimize for different criteria
   - Load balancing over equal-cost paths
5. **Authentication**:
   - LSPs can be authenticated

- Prevents malicious routing updates

Disadvantages of Link State:

1. **Memory Requirements**:
   - Must store complete topology database
   - $O(n^2)$ space for n routers
2. **CPU Intensive**:
   - Dijkstra algorithm complexity $O(n^2)$ or $O(n \log n)$
   - Must recalculate on topology changes
3. **Bandwidth for Flooding**:
   - Initial database synchronization
   - LSP floods during topology changes
4. **Complexity**:
   - More complex protocol implementation
   - Harder to debug

OSPF (Open Shortest Path First):

OSPF Hierarchy:

- **Areas**: Logical grouping of routers
- **Backbone Area (Area 0)**: Central area, all other areas connect to it
- **Area Border Routers (ABR)**: Connect areas
- **Autonomous System Boundary Routers (ASBR)**: Connect to other ASes

OSPF LSA Types:

1. **Router LSA (Type 1)**: Router's links within area
2. **Network LSA (Type 2)**: Multi-access network information
3. **Summary LSA (Type 3)**: Inter-area routes
4. **ASBR Summary LSA (Type 4)**: Location of ASBR
5. **External LSA (Type 5)**: External routes

OSPF Neighbor States:

1. **Down**: No Hello received
2. **Init**: Hello received, but not bidirectional
3. **2-Way**: Bidirectional communication established
4. **ExStart**: Master/slave relationship established
5. **Exchange**: Database description packets exchanged
6. **Loading**: Link state request/update packets exchanged
7. **Full**: Databases synchronized

Link State vs Distance Vector Summary:

Use Link State When:

- Fast convergence required
- Network topology changes frequently
- Loop-free routing critical
- Sufficient memory and CPU available
- Hierarchical design needed

Use Distance Vector When:

- Simple implementation preferred
- Limited resources (memory, CPU)
- Small, stable networks
- Minimal configuration desired

GATE Tips:

- Link state routers maintain complete topology database
- Uses Dijkstra's algorithm for shortest path calculation
- LSPs flooded reliably with sequence numbers and aging
- Faster convergence than distance vector (no count-to-infinity)
- Higher memory and CPU requirements
- OSPF is main link state protocol
- Event-driven updates (not periodic)
- Loop-free due to complete topology knowledge
- Scalable with hierarchical design (OSPF areas)

Flooding Routing Algorithm

Key Concepts: Flooding sends packets along all possible paths to guarantee delivery. Simple but inefficient algorithm used in specific scenarios.

Flooding Overview:

- **Principle**: Send packet copy on every outgoing link except arrival link
- **Guarantee**: If path exists, packet will reach destination
- **Efficiency**: Very low (generates many duplicate packets)
- **Use Cases**: Military networks, sensor networks, link state protocol updates

Basic Flooding Algorithm:

```
FloodingForward(packet, arrival_interface):
    if packet.destination == my_address:
```

```
        deliver_to_application(packet)
        return

    for each interface in outgoing_interfaces:
        if interface != arrival_interface:
            send(packet, interface)
```

Flooding Example:

Network topology:

```
    A -------- B
    |          |
    |          |
    C -------- D
```

Packet from A to D:

1. A sends to B and C
2. B receives from A, sends to D
3. C receives from A, sends to D
4. D receives packet twice (from B and C)

Problems with Basic Flooding:

1. Infinite Loops:

- Packets circulate forever in cycles
- Network becomes congested with duplicate packets

2. Packet Explosion:

- Number of packets grows exponentially
- In network with cycles, packets multiply indefinitely

3. Resource Waste:

- Bandwidth consumed by duplicate packets
- Processing overhead at each router

Flooding Control Mechanisms:

1. Hop Count (TTL - Time To Live):

- Each packet has maximum hop count
- Decremented at each router
- Packet discarded when hop count reaches 0

- **Problem**: Must estimate network diameter

```
FloodingWithTTL(packet, arrival_interface):
    packet.ttl = packet.ttl - 1
    if packet.ttl <= 0:
        discard(packet)
        return

    if packet.destination == my_address:
        deliver_to_application(packet)
        return

    for each interface in outgoing_interfaces:
        if interface != arrival_interface:
            send(packet, interface)
```

2. Sequence Numbers:

- Each source maintains sequence number
- Routers track (source, sequence) pairs seen
- Discard duplicates based on sequence number
- **Advantage**: Prevents loops completely
- **Disadvantage**: Requires state at each router

```
FloodingWithSequence(packet, arrival_interface):
    key = (packet.source, packet.sequence)
    if key in seen_packets:
        discard(packet)  // Duplicate
        return

    seen_packets.add(key)

    if packet.destination == my_address:
        deliver_to_application(packet)
        return

    for each interface in outgoing_interfaces:
        if interface != arrival_interface:
            send(packet, interface)
```

3. Reverse Path Forwarding (RPF):

- Accept packet only if it arrives on interface toward source
- Use routing table to determine "best" path to source
- Forward only if packet came from that direction
- **Advantage**: Reduces duplicates significantly

- **Disadvantage**: May not work with asymmetric routing

```
RPFFlooding(packet, arrival_interface):
    best_interface = routing_table.lookup(packet.source)
    if arrival_interface != best_interface:
        discard(packet)  // Wrong direction
        return

    if packet.destination == my_address:
        deliver_to_application(packet)
        return

    for each interface in outgoing_interfaces:
        if interface != arrival_interface:
            send(packet, interface)
```

Selective Flooding:

Concept: Flood only in "useful" directions

- Don't flood back toward source
- Don't flood to networks that don't contain destination
- Requires some topology knowledge

Implementation:

- Maintain partial routing information
- Flood only to interfaces that could lead to destination
- Balance between efficiency and simplicity

Flooding Applications:

1. Link State Routing Protocols:

- OSPF floods Link State Advertisements (LSAs)
- Ensures all routers have consistent topology database
- Uses sequence numbers and aging for control

2. Broadcast in LANs:

- Ethernet broadcasts use flooding within LAN segment
- Switches flood unknown unicast frames

3. Wireless Sensor Networks:

- Simple nodes with limited processing
- Flooding ensures message delivery in dynamic topology

- Energy efficiency more important than bandwidth

4. Military/Emergency Networks:

- Robustness more important than efficiency
- Network topology may be unknown or changing rapidly
- Flooding guarantees delivery if any path exists

5. Network Discovery:

- ARP requests flooded within broadcast domain
- DHCP discover messages flooded to find servers

Flooding Variants:

1. Controlled Flooding:

- Combine multiple control mechanisms
- Example: TTL + sequence numbers
- Better performance than basic flooding

2. Probabilistic Flooding:

- Forward packet with probability $p < 1$
- Reduces packet explosion
- May not guarantee delivery

3. Gossip Protocols:

- Forward to random subset of neighbors
- Used in distributed systems for information dissemination
- Balance between flooding and unicast

Performance Analysis:

Packet Complexity:

- **Basic Flooding**: O(E) packets per message (E = number of edges)
- **With Cycles**: Exponential growth without control
- **With Control**: O(E) packets maximum

Delivery Guarantee:

- **Basic Flooding**: 100% if any path exists
- **With TTL**: Depends on TTL value and network diameter
- **With RPF**: High probability, not guaranteed

Bandwidth Usage:

- **Worst Case**: Every link carries packet
- **Typical**: Much higher than optimal routing
- **Acceptable**: When reliability more important than efficiency

Flooding vs Other Routing:

Algorithm	Delivery	Efficiency	State Required	Convergence
Flooding	Guaranteed	Very Low	Minimal	Instant
Distance Vector	High	Medium	Medium	Slow
Link State	High	High	High	Fast
Source Routing	High	High	None	N/A

GATE Tips:

- Flooding sends packet on all outgoing links except arrival link
- Guarantees delivery if any path exists
- Problems: infinite loops, packet explosion, resource waste
- Control mechanisms: TTL, sequence numbers, reverse path forwarding
- Used in link state protocols (OSPF LSA flooding)
- Applications: broadcast, sensor networks, military networks
- Trade-off: reliability vs efficiency
- Packet complexity O(E) where E is number of edges
- Not practical for regular data forwarding due to inefficiency

Virtual Circuit Switching

Key Concepts: Virtual circuits establish dedicated logical paths through network before data transmission. Combines benefits of circuit and packet switching.

Virtual Circuit Overview:

- **Concept**: Logical connection established before data transfer
- **Path**: Fixed route through network for duration of connection
- **State**: Intermediate nodes maintain connection state
- **Phases**: Connection setup, data transfer, connection teardown

Circuit Switching vs Packet Switching vs Virtual Circuit:

Feature	Circuit Switching	Packet Switching	Virtual Circuit
Connection	Physical circuit	Connectionless	Logical circuit
Setup	Required	None	Required

Feature	Circuit Switching	Packet Switching	Virtual Circuit
State	End-to-end	None	Per-hop
Addressing	Once (setup)	Every packet	Label/VCI
Routing	Fixed path	Per-packet	Fixed path
Guarantees	Bandwidth, delay	None	Possible QoS
Efficiency	Low (unused bandwidth)	High	Medium

Virtual Circuit Components:

1. Virtual Circuit Identifier (VCI):

- **Purpose**: Label identifying virtual circuit
- **Scope**: Local significance (per-link)
- **Size**: Typically 12-16 bits
- **Mapping**: Different VCI on each link of path

2. Virtual Circuit Table:

- **Location**: Each intermediate router
- **Contents**: (Input Interface, Input VCI) → (Output Interface, Output VCI)
- **Function**: Maps incoming VC to outgoing VC

3. Signaling Protocol:

- **Purpose**: Establish and tear down virtual circuits
- **Messages**: Setup, Connect, Disconnect, Release
- **Examples**: Q.931 (ISDN), Q.2931 (ATM)

Virtual Circuit Establishment:

Setup Phase:

1. **Setup Request**: Source sends setup message with destination address
2. **Path Selection**: Each router chooses next hop toward destination
3. **VCI Allocation**: Each router allocates VCI for this connection
4. **Table Entry**: Each router creates forwarding table entry
5. **Setup Confirmation**: Destination sends confirmation back to source

Example Setup:

```
Network: A --- R1 --- R2 --- R3 --- B

Setup Process:
1. A sends Setup(dest=B) to R1
```

```
2. R1 allocates VCI=5, forwards Setup to R2
3. R2 allocates VCI=8, forwards Setup to R3
4. R3 allocates VCI=3, forwards Setup to B
5. B sends Connect message back
6. Each router creates table entry during setup
```

Virtual Circuit Tables After Setup:

Router R1:
Input Interface	Input VCI	Output Interface	Output VCI
A	5	R2	8

Router R2:
Input Interface	Input VCI	Output Interface	Output VCI
R1	8	R3	3

Data Transfer Phase:

Packet Format:

```
+--------+--------+------------------+
|  VCI   | Length |      Data        |
+--------+--------+------------------+
```

Forwarding Process:

1. Packet arrives with VCI
2. Router looks up (Input Interface, Input VCI) in table
3. Router replaces VCI with Output VCI
4. Router forwards packet on Output Interface

Example Data Transfer:

```
A sends packet with VCI=5 to R1
R1 receives on interface A with VCI=5
R1 looks up table: (A,5) → (R2,8)
R1 changes VCI to 8, forwards to R2
R2 receives on interface R1 with VCI=8
R2 looks up table: (R1,8) → (R3,3)
R2 changes VCI to 3, forwards to R3
...continues to destination B
```

Connection Teardown:

1. Either end sends Disconnect message

2. Message follows same path as data
3. Each router removes table entry
4. VCIs are deallocated for reuse
5. Resources freed

Virtual Circuit Types:

1. Switched Virtual Circuits (SVC):

- **Dynamic**: Established on-demand
- **Temporary**: Exist only for duration of communication
- **Signaling**: Requires signaling protocol
- **Example**: ATM SVCs, Frame Relay SVCs

2. Permanent Virtual Circuits (PVC):

- **Static**: Pre-configured by administrator
- **Persistent**: Always available
- **No Signaling**: No setup/teardown messages
- **Example**: Leased lines, ATM PVCs, Frame Relay PVCs

Advantages of Virtual Circuits:

1. Quality of Service (QoS):

- Resources can be reserved during setup
- Bandwidth, delay, jitter guarantees possible
- Traffic shaping and policing

2. Simplified Forwarding:

- Fast table lookup using VCI
- No complex routing decisions per packet
- Hardware-friendly implementation

3. Connection State:

- Network maintains connection information
- Error detection and recovery possible
- Flow control between adjacent nodes

4. Addressing Efficiency:

- Short VCI instead of full destination address
- Reduces packet header overhead

5. Traffic Engineering:

- Explicit path selection possible
- Load balancing across multiple paths
- Network resource optimization

Disadvantages of Virtual Circuits:

1. Setup Overhead:

- Connection establishment delay
- Signaling protocol complexity
- Not efficient for short communications

2. State Maintenance:

- Routers must maintain per-connection state
- Memory overhead
- State consistency issues

3. Failure Handling:

- Connection state lost if router fails
- Must re-establish connections
- More complex than stateless packet switching

4. Scalability:

- Limited by number of VCIs
- State storage requirements grow with connections
- Signaling load increases with connection rate

Virtual Circuit Technologies:

1. ATM (Asynchronous Transfer Mode):

- **Cell Size**: Fixed 53 bytes (5 header + 48 payload)
- **VPI/VCI**: Virtual Path/Virtual Channel Identifiers
- **QoS**: Multiple service categories (CBR, VBR, ABR, UBR)
- **Applications**: Backbone networks, multimedia

2. Frame Relay:

- **Variable Frames**: Up to 4096 bytes
- **DLCI**: Data Link Connection Identifier
- **CIR**: Committed Information Rate
- **Applications**: WAN connectivity, legacy networks

3. MPLS (Multi-Protocol Label Switching):

- **Labels**: 20-bit label identifies forwarding equivalence class
- **LSP**: Label Switched Path (similar to virtual circuit)
- **Traffic Engineering**: Explicit routing capabilities
- **Applications**: Service provider networks, VPNs

Virtual Circuit Implementation:

Connection Table Structure:

```
struct VCEntry {
    int input_interface;
    int input_vci;
    int output_interface;
    int output_vci;
    QoSParameters qos;
    Statistics stats;
};

class VirtualCircuitSwitch {
    map<pair<int,int>, VCEntry> vc_table;  // (interface, vci) -> entry

    void forward_packet(Packet& pkt, int input_if) {
        auto key = make_pair(input_if, pkt.vci);
        if (vc_table.find(key) != vc_table.end()) {
            VCEntry& entry = vc_table[key];
            pkt.vci = entry.output_vci;
            send_packet(pkt, entry.output_interface);
            update_statistics(entry);
        } else {
            drop_packet(pkt);  // No VC found
        }
    }
};
```

Performance Considerations:

Forwarding Speed:

- Hash table lookup: O(1) average case
- Hardware implementation possible
- Faster than IP routing table lookup

Memory Usage:

- Per-connection state: ~100 bytes per VC
- 1M connections ≈ 100 MB memory
- Scalability limited by memory

Setup Time:

- Round-trip time for setup
- Processing delay at each hop
- Acceptable for long-lived connections

GATE Tips:

- Virtual circuits establish logical connection before data transfer
- VCI (Virtual Circuit Identifier) labels packets
- VCI has local significance (different on each link)
- Three phases: setup, data transfer, teardown
- Routers maintain per-connection state in VC tables
- Advantages: QoS support, fast forwarding, traffic engineering
- Disadvantages: setup overhead, state maintenance, failure complexity
- Examples: ATM, Frame Relay, MPLS
- SVC (dynamic) vs PVC (permanent) virtual circuits
- Combines benefits of circuit and packet switching

CIDR (Classless Inter-Domain Routing)

Key Concepts: CIDR eliminates traditional IP address classes, enabling flexible subnet allocation and route aggregation. Essential for IPv4 address conservation and routing scalability.

CIDR Overview:

- **Purpose**: Replace classful addressing with flexible prefix lengths
- **Notation**: IP address followed by prefix length (e.g., 192.168.1.0/24)
- **Benefits**: Efficient address allocation, route aggregation, reduced routing table size
- **Deployment**: Mandatory in modern Internet (since 1993)

Classful vs Classless Addressing:

Classful Addressing (Legacy):

- **Class A**: 1.0.0.0 to 126.255.255.255 (/8 networks)
  - Network: 8 bits, Host: 24 bits
  - 126 networks, 16.7M hosts each
- **Class B**: 128.0.0.0 to 191.255.255.255 (/16 networks)
  - Network: 16 bits, Host: 16 bits
  - 16,384 networks, 65,534 hosts each
- **Class C**: 192.0.0.0 to 223.255.255.255 (/24 networks)
  - Network: 24 bits, Host: 8 bits

- 2.1M networks, 254 hosts each

Problems with Classful:

1. **Address Waste**: Organization needing 1000 hosts gets Class B (65,534 hosts)
2. **Routing Table Growth**: Each network needs separate route
3. **Inflexibility**: Fixed network sizes don't match requirements

CIDR Addressing:

- **Variable Length**: Network prefix can be any length (1-30 bits)
- **Notation**: Network/prefix_length (e.g., 203.0.113.0/25)
- **Subnet Mask**: Derived from prefix length
- **Flexibility**: Allocate exactly what's needed

CIDR Notation Examples:

CIDR	Subnet Mask	Network Bits	Host Bits	Addresses	Hosts
/8	255.0.0.0	8	24	16,777,216	16,777,214
/16	255.255.0.0	16	16	65,536	65,534
/24	255.255.255.0	24	8	256	254
/25	255.255.255.128	25	7	128	126
/26	255.255.255.192	26	6	64	62
/27	255.255.255.224	27	5	32	30
/28	255.255.255.240	28	4	16	14
/30	255.255.255.252	30	2	4	2

CIDR Calculations:

Given CIDR Block 192.168.1.0/26:

Network Information:

- **Network Address**: 192.168.1.0
- **Prefix Length**: 26 bits
- **Subnet Mask**: 255.255.255.192 (11111111.11111111.11111111.11000000)
- **Host Bits**: 32 - 26 = 6 bits
- **Total Addresses**: 2^6 = 64
- **Usable Hosts**: 64 - 2 = 62 (exclude network and broadcast)

Address Range:

- **First Address**: 192.168.1.0 (network address)

- **First Host**: 192.168.1.1
- **Last Host**: 192.168.1.62
- **Broadcast**: 192.168.1.63

Binary Analysis:

```
Network: 192.168.1.0/26
Binary:  11000000.10101000.00000001.00000000
Mask:    11111111.11111111.11111111.11000000
         |-------- Network (26 bits) -------|Host|
```

Subnetting with CIDR:

Example: Subnet 192.168.1.0/24 into smaller networks

Requirements:

- Subnet A: 50 hosts
- Subnet B: 25 hosts
- Subnet C: 10 hosts
- Subnet D: 5 hosts

Solution:

1. **Subnet A**: Needs 50 hosts → 6 host bits (2^6 = 64) → /26
   - Network: 192.168.1.0/26 (192.168.1.0 - 192.168.1.63)
2. **Subnet B**: Needs 25 hosts → 5 host bits (2^5 = 32) → /27
   - Network: 192.168.1.64/27 (192.168.1.64 - 192.168.1.95)
3. **Subnet C**: Needs 10 hosts → 4 host bits (2^4 = 16) → /28
   - Network: 192.168.1.96/28 (192.168.1.96 - 192.168.1.111)
4. **Subnet D**: Needs 5 hosts → 3 host bits (2^3 = 8) → /29
   - Network: 192.168.1.112/29 (192.168.1.112 - 192.168.1.119)

Route Aggregation (Supernetting):

Concept: Combine multiple networks into single route entry

Example: Aggregate these networks:

- 203.0.113.0/24
- 203.0.114.0/24
- 203.0.115.0/24
- 203.0.116.0/24

Binary Analysis:

```
203.0.113.0: 11001011.00000000.01110001.00000000
203.0.114.0: 11001011.00000000.01110010.00000000
203.0.115.0: 11001011.00000000.01110011.00000000
203.0.116.0: 11001011.00000000.01110100.00000000
             |----Common Prefix----|
```

Common Prefix: 22 bits (11001011.00000000.011100)
Aggregate Route: 203.0.112.0/22

Verification:

- Covers: 203.0.112.0 to 203.0.115.255
- Includes all four /24 networks
- Single route entry instead of four

VLSM (Variable Length Subnet Masking):

Concept: Use different subnet mask lengths within same network

Example: 172.16.0.0/16 network with VLSM

Subnets:

- **WAN Links**: /30 (2 hosts each)
  - 172.16.1.0/30, 172.16.1.4/30, 172.16.1.8/30
- **Small LANs**: /27 (30 hosts each)
  - 172.16.2.0/27, 172.16.2.32/27, 172.16.2.64/27
- **Large LANs**: /24 (254 hosts each)
  - 172.16.10.0/24, 172.16.11.0/24

Benefits:

- **Efficient**: Right-size subnets for requirements
- **Scalable**: Hierarchical addressing
- **Flexible**: Mix different subnet sizes

CIDR and Routing:

Longest Prefix Match:

- Router chooses most specific route (longest prefix)
- Enables hierarchical routing and aggregation

Example Routing Table:
Destination	Next Hop	Interface
0.0.0.0/0	10.1.1.1	eth0	(Default route)

| 192.168.0.0/16 | 10.1.1.2 | eth1 | (Aggregate)
| 192.168.1.0/24 | 10.1.1.3 | eth2 | (Specific)
| 192.168.1.128/25 | 10.1.1.4 | eth3 | (More specific)

Packet to 192.168.1.200:

- Matches: 0.0.0.0/0, 192.168.0.0/16, 192.168.1.0/24
- **Longest match**: 192.168.1.0/24 → Forward via 10.1.1.3

Packet to 192.168.1.150:

- Matches: 0.0.0.0/0, 192.168.0.0/16, 192.168.1.0/24, 192.168.1.128/25
- **Longest match**: 192.168.1.128/25 → Forward via 10.1.1.4

CIDR Benefits:

1. Address Efficiency:

- Allocate exactly needed addresses
- Reduce address waste
- Extend IPv4 lifetime

2. Routing Scalability:

- Route aggregation reduces table size
- Hierarchical addressing structure
- Faster routing lookups

3. Flexibility:

- Variable subnet sizes
- Easy network renumbering
- Support for different requirements

4. ISP Allocation:

- ISPs get large blocks, subdivide efficiently
- Customers get right-sized allocations
- Simplified address management

CIDR Allocation Hierarchy:

Internet Assigned Numbers Authority (IANA):

- Allocates large blocks to Regional Internet Registries (RIRs)

Regional Internet Registries:

- **ARIN**: North America (e.g., 24.0.0.0/8)
- **RIPE**: Europe (e.g., 80.0.0.0/8)
- **APNIC**: Asia-Pacific (e.g., 202.0.0.0/8)
- **LACNIC**: Latin America
- **AFRINIC**: Africa

ISPs and Organizations:

- Receive allocations from RIRs
- Further subdivide for customers
- Maintain hierarchical structure

CIDR Tools and Calculations:

Subnet Calculator:

```
def cidr_info(network, prefix_len):
    # Convert to binary
    net_int = ip_to_int(network)

    # Calculate mask
    mask = (0xFFFFFFFF << (32 - prefix_len)) & 0xFFFFFFFF

    # Network address
    net_addr = net_int & mask

    # Broadcast address
    broadcast = net_addr | (0xFFFFFFFF >> prefix_len)

    # Host range
    first_host = net_addr + 1
    last_host = broadcast - 1

    # Number of hosts
    num_hosts = (1 << (32 - prefix_len)) - 2

    return {
        'network': int_to_ip(net_addr),
        'broadcast': int_to_ip(broadcast),
        'first_host': int_to_ip(first_host),
        'last_host': int_to_ip(last_host),
        'num_hosts': num_hosts,
        'subnet_mask': int_to_ip(mask)
    }
```

Common CIDR Blocks:

- **/8**: 16.7M addresses (Class A equivalent)
- **/16**: 65K addresses (Class B equivalent)
- **/24**: 256 addresses (Class C equivalent)
- **/25**: 128 addresses (half of /24)
- **/26**: 64 addresses (quarter of /24)
- **/27**: 32 addresses
- **/28**: 16 addresses
- **/29**: 8 addresses
- **/30**: 4 addresses (point-to-point links)

GATE Tips:

- CIDR uses variable-length prefixes instead of fixed classes
- Notation: network/prefix_length (e.g., 192.168.1.0/24)
- Number of hosts = 2^(32-prefix_length) - 2
- Subnet mask derived from prefix length
- Route aggregation combines multiple networks into single route
- Longest prefix match used for routing decisions
- VLSM allows different subnet sizes within same network
- Benefits: address efficiency, routing scalability, flexibility
- Essential for modern Internet routing and addressing

Missing Topics - Compiler Design

Constant Propagation

Key Concepts: Constant propagation is a compiler optimization that replaces variables with their constant values when possible. Essential data flow analysis technique for code optimization.

Constant Propagation Overview:

- **Purpose**: Replace variable references with known constant values
- **Type**: Forward data flow analysis
- **Goal**: Eliminate unnecessary variable loads and enable further optimizations
- **Phase**: Typically performed during optimization phase

Basic Concept:

```
// Before constant propagation
int x = 5;
int y = x + 3;
int z = y * 2;
return z;
```

```
// After constant propagation
int x = 5;
int y = 5 + 3;    // x replaced with 5
int z = 8 * 2;    // y replaced with 8
return 16;        // z replaced with 16
```

Data Flow Analysis Framework:

Domain: Each variable can be in one of three states:

- $\perp$ **(Bottom)**: Undefined/uninitialized
- **Constant c**: Variable has definite constant value c
- $\top$ **(Top)**: Variable has unknown/non-constant value

Lattice Structure:

```
        ⊤ (Non-constant)
      / | \
     /  |  \
   c₁  c₂  c₃  ... (All possible constants)
     \  |  /
      \ | /
        ⊥ (Undefined)
```

Meet Operation ($\wedge$):

- $\perp \wedge$ **x = x** (undefined meets anything = that thing)
- **c** $\wedge$ **c = c** (same constant meets itself = same constant)
- $\mathbf{c_1} \wedge \mathbf{c_2} = \top$ (different constants meet = non-constant)
- $\top \wedge$ **x =** $\top$ (non-constant meets anything = non-constant)

Transfer Functions:

Assignment Statements:

1. **x = c** (constant assignment):
   - OUT[x] = c
   - All other variables unchanged
2. **x = y** (copy assignment):
   - OUT[x] = IN[y]
   - All other variables unchanged
3. **x = y op z** (binary operation):
   - If IN[y] = c_1 and IN[z] = c_2 (both constants):
     - OUT[x] = c_1 op c_2 (if computable)

- Otherwise: OUT[x] = ⊤

4. **x = f(...)** (function call):
   - OUT[x] = ⊤ (conservative assumption)

Control Flow Handling:

Basic Blocks: Apply transfer functions sequentially

```
IN[s₁] = IN[block]
OUT[s₁] = transfer(s₁, IN[s₁])
IN[s₂] = OUT[s₁]
OUT[s₂] = transfer(s₂, IN[s₂])
...
OUT[block] = OUT[sₙ]
```

Join Points: Merge information from multiple predecessors

```
IN[block] = ∧ OUT[pred] for all predecessors pred
```

Algorithm:

Iterative Data Flow Analysis:

```
ConstantPropagation(CFG):
    // Initialize
    for each variable v:
        IN[entry][v] = ⊥
        for each block B ≠ entry:
            IN[B][v] = ⊥

    // Iterate until convergence
    changed = true
    while changed:
        changed = false
        for each block B in CFG:
            // Compute IN[B] from predecessors
            old_in = IN[B]
            IN[B] = ∧ OUT[pred] for all pred of B

            // Apply transfer function
            old_out = OUT[B]
            OUT[B] = transfer(B, IN[B])

            if IN[B] ≠ old_in or OUT[B] ≠ old_out:
                changed = true

    return IN, OUT
```

Example Analysis:

Code:

```
1:  x = 5;
2:  y = x + 2;
3:  if (condition)
4:      z = y * 3;
5:  else
6:      z = 10;
7:  w = z + 1;
```

Control Flow Graph:

Data Flow Analysis:

Block [1,2,3]:

- IN: x=⊥, y=⊥, z=⊥, w=⊥
- After x=5: x=5, y=⊥, z=⊥, w=⊥
- After y=x+2: x=5, y=7, z=⊥, w=⊥
- OUT: x=5, y=7, z=⊥, w=⊥

Block [4]:

- IN: x=5, y=7, z=⊥, w=⊥
- After z=y*3: x=5, y=7, z=21, w=⊥
- OUT: x=5, y=7, z=21, w=⊥

Block [6]:

- IN: x=5, y=7, z=⊥, w=⊥
- After z=10: x=5, y=7, z=10, w=⊥
- OUT: x=5, y=7, z=10, w=⊥

Block [7] (Join point):

- IN: x=5, y=7, z=(21∧10)=⊤, w=⊥
- After w=z+1: x=5, y=7, z=⊤, w=⊤

- OUT: x=5, y=7, z=⊤, w=⊤

Optimized Code:

```
1:  x = 5;
2:  y = 7;          // x+2 = 5+2 = 7
3:  if (condition)
4:      z = 21;     // y*3 = 7*3 = 21
5:  else
6:      z = 10;
7:  w = z + 1;      // z is non-constant, can't optimize
```

Advanced Constant Propagation:

Conditional Constant Propagation:

- Consider branch conditions in analysis
- Propagate constants along feasible paths only

Example:

```
x = 5;
if (x > 3) {     // Always true since x = 5
    y = 10;
} else {
    y = 20;      // Dead code
}
z = y + 1;       // y = 10, so z = 11
```

Sparse Constant Propagation:

- Use SSA (Static Single Assignment) form
- More efficient than dense analysis
- Propagate constants through def-use chains

Interprocedural Constant Propagation:

- Analyze across function boundaries
- Handle parameter passing and return values
- More complex but more effective

Implementation Considerations:

Representation:

```
enum ConstantValue {
    BOTTOM,      // Undefined
```

```
    TOP,        // Non-constant
    CONSTANT    // Has constant value
};

struct ConstInfo {
    ConstantValue type;
    int value;  // Only valid if type == CONSTANT
};

class ConstantPropagation {
    map<Variable, ConstInfo> constants;

    ConstInfo meet(ConstInfo a, ConstInfo b) {
        if (a.type == BOTTOM) return b;
        if (b.type == BOTTOM) return a;
        if (a.type == TOP || b.type == TOP)
            return {TOP, 0};
        if (a.value == b.value)
            return a;
        return {TOP, 0};
    }

    ConstInfo transfer(Instruction inst, map<Variable, ConstInfo>& in) {
        switch (inst.type) {
            case ASSIGN_CONST:
                return {CONSTANT, inst.value};
            case ASSIGN_VAR:
                return in[inst.src];
            case BINARY_OP:
                if (in[inst.left].type == CONSTANT &&
                    in[inst.right].type == CONSTANT) {
                    int result = evaluate(inst.op,
                                          in[inst.left].value,
                                          in[inst.right].value);
                    return {CONSTANT, result};
                }
                return {TOP, 0};
            default:
                return {TOP, 0};
        }
    }
};
```

Benefits of Constant Propagation:

1. **Direct Benefits**:
   - Eliminate variable loads
   - Replace with immediate values

- Reduce register pressure

2. **Enable Other Optimizations**:
   - **Constant Folding**: Evaluate constant expressions at compile time
   - **Dead Code Elimination**: Remove unreachable branches
   - **Strength Reduction**: Replace expensive operations with cheaper ones
   - **Loop Optimizations**: Unroll loops with constant bounds
3. **Performance Impact**:
   - Faster execution (fewer memory accesses)
   - Smaller code size (immediate values vs. loads)
   - Better cache performance

Limitations:

1. **Aliasing**: Pointers can create uncertainty

```
int x = 5;
int *p = &x;
*p = 10;      // x is no longer constant
int y = x;    // Can't propagate constant
```

2. **Function Calls**: May modify global variables

```
int x = 5;
foo();        // Might modify x
int y = x;    // Can't assume x is still 5
```

3. **Arrays**: Array elements difficult to track

```
int a[10];
a[0] = 5;
int i = 0;
int x = a[i];  // Hard to determine a[i] = a[0] = 5
```

Constant Propagation vs Related Optimizations:

Optimization	Purpose	When Applied
Constant Propagation	Replace variables with constants	Data flow analysis
Constant Folding	Evaluate constant expressions	During/after propagation
Dead Code Elimination	Remove unreachable code	After propagation
Copy Propagation	Replace x with y when x=y	Similar to constant prop

GATE Tips:

- Constant propagation replaces variables with known constant values
- Uses forward data flow analysis with lattice: $\perp \rightarrow$ constants $\rightarrow \top$

- Meet operation: same constants merge to same, different merge to $\top$
- Transfer functions handle assignments, operations, and control flow
- Enables other optimizations: constant folding, dead code elimination
- Limitations: aliasing, function calls, arrays create uncertainty
- Iterative algorithm converges due to monotonic lattice
- More effective in SSA form (sparse constant propagation)
- Essential optimization for performance improvement

Missing Topics - Database Management Systems

File Organization

Key Concepts: File organization determines how records are physically stored and accessed on disk. Critical for database performance and storage efficiency.

File Organization Overview:

- **Purpose**: Organize records for efficient storage and retrieval
- **Trade-offs**: Space utilization vs. access speed vs. update cost
- **Factors**: Access patterns, record size, update frequency, storage constraints
- **Impact**: Directly affects query performance and storage requirements

Types of File Organization:

1. Heap Files (Unordered Files)

Characteristics:

- **Structure**: Records stored in no particular order
- **Insertion**: New records added at end of file
- **Storage**: Simplest organization method
- **Use Case**: When no specific access pattern dominates

Operations:

Insert: O(1)

- Add new record at end of file
- Update file header with new record count
- Most efficient insertion method

Search: O(n)

- Linear scan through entire file
- Must examine every record in worst case
- No optimization possible without additional structures

Delete: O(n) + reorganization cost

- Find record (linear search)
- Mark as deleted or physically remove
- May leave gaps requiring periodic reorganization

Update: O(n)

- Find record (linear search)
- Modify in place if size unchanged
- If size changes, may need to relocate

Advantages:

- Simple implementation
- Fast insertion
- No overhead for maintaining order
- Good when file is small or access is random

Disadvantages:

- Slow search and retrieval
- Inefficient for range queries
- Wasted space from deleted records
- Poor performance for large files

Example Structure:

```
File Header: [Record Count][Free Space Pointer]
Record 1: [ID=101][Name=Alice][Age=25]
Record 2: [ID=205][Name=Bob][Age=30]
Record 3: [ID=150][Name=Carol][Age=28]
...
Free Space: [Available for new records]
```

2. Sorted Files (Sequential Files)

Characteristics:

- **Structure**: Records sorted by key field(s)
- **Ordering**: Maintained physically on disk
- **Access**: Efficient for range queries and sequential access
- **Maintenance**: Expensive to maintain sort order

Operations:

Search: O(log n) using binary search

- Divide and conquer approach
- Much faster than linear search
- Efficient for exact match and range queries

Insert: O(n)

- Find correct position (binary search)
- Shift records to make space
- Insert new record
- Expensive due to record shifting

Delete: O(log n) + O(n) for reorganization

- Find record using binary search
- Remove record
- Shift remaining records to fill gap

Update: O(log n) if key unchanged, O(n) if key changes

- Non-key updates: find and modify in place
- Key updates: delete old, insert new (expensive)

Advantages:

- Fast search using binary search
- Excellent for range queries
- Sequential access is very efficient
- Good for read-heavy workloads

Disadvantages:

- Expensive insertions and deletions
- Requires periodic reorganization
- Poor for write-heavy workloads
- Overflow handling complex

Example Structure:

```
Record 1: [ID=101][Name=Alice][Age=25]
Record 2: [ID=150][Name=Carol][Age=28]
Record 3: [ID=205][Name=Bob][Age=30]
Record 4: [ID=301][Name=Dave][Age=35]
...
(Sorted by ID field)
```

Binary Search Implementation:

```
int binarySearch(File& file, int key) {
    int left = 0, right = file.recordCount - 1;

    while (left <= right) {
        int mid = (left + right) / 2;
        Record record = file.readRecord(mid);

        if (record.key == key)
            return mid;
        else if (record.key < key)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1; // Not found
}
```

3. Hash Files

Characteristics:

- **Structure**: Records distributed using hash function
- **Access**: Direct access using hash key
- **Performance**: O(1) average case for exact match
- **Collision Handling**: Various techniques to handle hash collisions

Hash Function:

- Maps key values to bucket addresses
- Should distribute records uniformly
- Common functions: division, multiplication, folding

Collision Resolution:

Open Addressing:

- **Linear Probing**: Check next sequential location
- **Quadratic Probing**: Check locations at quadratic intervals
- **Double Hashing**: Use second hash function for probe sequence

Separate Chaining:

- Each bucket contains linked list of records
- Handle collisions by adding to chain
- More flexible but requires pointer overhead

Operations:

Search: O(1) average, O(n) worst case

- Apply hash function to key
- Check bucket directly
- Handle collisions as needed

Insert: O(1) average

- Hash key to find bucket
- Insert if space available
- Handle collision if bucket full

Delete: O(1) average

- Find record using hash
- Remove from bucket
- May need to reorganize chain

Advantages:

- Very fast exact match queries
- Constant time operations (average case)
- Good for equality-based access
- Suitable for high-volume transactions

Disadvantages:

- Poor for range queries
- Hash function quality critical
- Collision handling overhead
- Difficult to maintain high load factor

Example Hash File:

```
Hash Function: h(key) = key % 7

Bucket 0: [ID=105][Name=Eve][Age=22]
Bucket 1: [ID=101][Name=Alice][Age=25] → [ID=108][Name=Frank][Age=40]
Bucket 2: [ID=205][Name=Bob][Age=30]
Bucket 3: [ID=150][Name=Carol][Age=28]
Bucket 4: [Empty]
Bucket 5: [ID=301][Name=Dave][Age=35]
Bucket 6: [Empty]
```

Dynamic Hashing:

- **Extendible Hashing**: Directory-based approach, grows dynamically
- **Linear Hashing**: Splits buckets incrementally
- **Advantages**: Handles growing files efficiently
- **Complexity**: More complex implementation

4. Clustered Files

Characteristics:

- **Structure**: Records with related key values stored together
- **Clustering**: Based on clustering key (may be non-unique)
- **Access**: Efficient for queries on clustering key
- **Organization**: Combines aspects of sorted and hash organization

Types:

Index Clustering:

- Records with same key value stored in same block
- Index points to first record of each key value
- Efficient for queries with equality on clustering key

Hash Clustering:

- Use hash function on clustering key
- Records with same hash value stored together
- Good for equality queries, poor for range queries

Operations:

- **Search**: Efficient for clustering key queries
- **Insert**: May require finding appropriate cluster
- **Range Queries**: Efficient if clustering key used in range
- **Updates**: Efficient if clustering key unchanged

Advantages:

- Reduces I/O for queries on clustering key
- Good for join operations
- Improves cache locality
- Reduces index size

Disadvantages:

- Poor performance for non-clustering key queries
- Complex maintenance
- May waste space if clusters uneven
- Insertion may be expensive

File Organization Comparison

Organization	Search	Insert	Delete	Range Query	Space Util	Best Use Case
Heap	O(n)	O(1)	O(n)	O(n)	Good	Random access, small files
Sorted	O(log n)	O(n)	O(n)	O(log n + k)	Good	Read-heavy, range queries
Hash	O(1) avg	O(1) avg	O(1) avg	O(n)	Fair	Exact match, high volume
Clustered	Varies	Varies	Varies	Good for cluster key	Fair	Related data access

Factors in Choosing File Organization:

1. **Access Patterns**:
   - Random vs. sequential access
   - Exact match vs. range queries
   - Read vs. write frequency
2. **Data Characteristics**:
   - File size and growth rate
   - Record size and variability
   - Key distribution and uniqueness
3. **Performance Requirements**:
   - Response time constraints
   - Throughput requirements
   - Concurrent access needs
4. **Storage Constraints**:
   - Available disk space
   - I/O bandwidth limitations
   - Memory availability

Modern Considerations:

SSD vs. HDD:

- SSDs reduce seek time penalty
- Random access becomes more viable
- May change optimal organization choice

Column Stores:

- Store columns separately instead of rows
- Better compression and cache performance
- Good for analytical workloads

Log-Structured Storage:

- Append-only writes for better performance
- Used in modern NoSQL databases
- Handles write-heavy workloads efficiently

GATE Tips:

- Heap files: unordered, O(1) insert, O(n) search
- Sorted files: ordered, O(log n) search, O(n) insert
- Hash files: O(1) average operations, poor for range queries
- Clustered files: group related records, good for clustering key queries
- Choice depends on access patterns and performance requirements
- Trade-offs between search speed, update cost, and space utilization
- Binary search works only on sorted files
- Hash functions should distribute records uniformly
- Consider workload characteristics when choosing organization

Missing Topics - Computer Organization

Ethernet Bridging

Key Concepts: Ethernet bridging connects multiple network segments at the data link layer, creating larger collision domains while maintaining separate broadcast domains per segment.

Bridge Overview:

- **Function**: Connect multiple Ethernet segments
- **Layer**: Data Link Layer (Layer 2)
- **Purpose**: Extend network reach, reduce collisions, filter traffic
- **Intelligence**: Learn MAC addresses, make forwarding decisions

Bridge vs. Hub vs. Switch:

Device	Layer	Collision Domains	Broadcast Domains	Intelligence
Hub	Physical	1 (all ports)	1 (all ports)	None (repeater)
Bridge	Data Link	1 per port	1 (all ports)	MAC learning
Switch	Data Link	1 per port	1 per VLAN	MAC learning + more

Bridge Operation:

Learning Process

MAC Address Learning:

1. **Initialize**: Bridge starts with empty MAC address table
2. **Receive Frame**: Frame arrives on port
3. **Learn Source**: Record source MAC address and incoming port
4. **Age Entries**: Remove old entries after timeout (typically 300 seconds)
5. **Update**: Refresh timestamp for known addresses

Learning Table Structure:

```
MAC Address    | Port | Age | Status
---------------|------|-----|--------
00:1A:2B:3C:4D | 1    | 45  | Active
00:2C:4D:5E:6F | 2    | 120 | Active
00:3E:5F:6A:7B | 1    | 200 | Active
```

Forwarding Process

Frame Processing Algorithm:

```
BridgeForward(frame, input_port):
    // Learn source address
    learn(frame.source_mac, input_port)

    // Check destination
    if frame.destination_mac == broadcast:
        flood_except(frame, input_port)
    else if frame.destination_mac in mac_table:
        output_port = mac_table[frame.destination_mac]
        if output_port != input_port:
            forward(frame, output_port)
        // else filter (same segment)
    else:
        flood_except(frame, input_port)  // Unknown destination
```

Forwarding Decisions:

1. **Forward**: Send frame to specific port (destination known, different segment)
2. **Filter**: Drop frame (destination on same segment as source)
3. **Flood**: Send to all ports except incoming port (broadcast or unknown destination)

Example Network:

```
Segment A ---- Bridge ---- Segment B
   |                          |
Host X                    Host Y
(MAC: AA)                 (MAC: BB)
```

Scenario 1 - Initial State:

- Bridge table empty
- Host X sends frame to Host Y
- Bridge learns X's MAC on Port A
- Destination unknown → floods to Port B
- Host Y receives frame

Scenario 2 - After Learning:

- Host Y replies to Host X
- Bridge learns Y's MAC on Port B
- Destination (X) known on Port A → forwards to Port A
- No flooding needed

Scenario 3 - Same Segment:

- Host X sends to another host on Segment A
- Bridge learns/refreshes X's MAC
- Destination on same segment → filters (doesn't forward)

Spanning Tree Protocol (STP)

Problem: Multiple bridges create loops

- Broadcast storms
- MAC table instability
- Frame duplication

Solution: Spanning Tree Protocol (IEEE 802.1D)

- Creates loop-free topology
- Maintains redundancy for fault tolerance

- Automatically reconfigures on failures

STP Concepts:

Root Bridge:

- Bridge with lowest Bridge ID
- Bridge ID = Priority (2 bytes) + MAC Address (6 bytes)
- Center of spanning tree

Port States:

1. **Blocking**: Receives BPDUs, doesn't forward data
2. **Listening**: Participates in STP, doesn't learn MACs
3. **Learning**: Learns MAC addresses, doesn't forward data
4. **Forwarding**: Normal operation, forwards data
5. **Disabled**: Port shut down

Port Roles:

- **Root Port**: Best path to root bridge
- **Designated Port**: Best path to segment
- **Blocked Port**: Alternate path (prevents loops)

BPDU (Bridge Protocol Data Unit):

- Messages exchanged between bridges
- Contains: Root ID, Bridge ID, Path Cost, Port ID
- Sent every 2 seconds (Hello Time)

STP Algorithm:

1. **Elect Root Bridge**: Lowest Bridge ID
2. **Calculate Root Path Cost**: Sum of link costs to root
3. **Elect Designated Bridge**: Lowest cost to root for each segment
4. **Block Redundant Ports**: Create loop-free topology

Example STP Topology:

```
     Bridge A (Root)
    /              \
 Bridge B        Bridge C
    \              /
     Bridge D ----
```

After STP:

- Bridge A: All ports forwarding (root)
- Bridge B: Port to A forwarding, port to D forwarding
- Bridge C: Port to A forwarding, port to D blocked
- Bridge D: Port to B forwarding, port to C blocked

Bridge Types

Transparent Bridge:

- **Operation**: Invisible to end stations
- **Learning**: Automatic MAC address learning
- **Standard**: IEEE 802.1D
- **Most Common**: Used in Ethernet LANs

Source Routing Bridge:

- **Operation**: End stations specify route
- **Learning**: Stations discover routes
- **Usage**: Token Ring networks (legacy)
- **Complexity**: Requires intelligent end stations

Translational Bridge:

- **Purpose**: Connect different LAN types
- **Function**: Translate between protocols (Ethernet ↔ Token Ring)
- **Challenges**: Different frame formats, addressing schemes
- **Usage**: Legacy network integration

Bridge Performance

Forwarding Rate:

- **Wire Speed**: Forward at full link speed
- **Factors**: Processing power, memory bandwidth, table lookup speed
- **Bottlenecks**: CPU, memory, backplane capacity

Latency:

- **Store-and-Forward**: Receive entire frame before forwarding
- **Cut-Through**: Start forwarding before complete reception
- **Trade-off**: Latency vs. error detection

Filtering Rate:

- Rate at which bridge can make forwarding decisions
- Important for high-traffic networks

- Measured in frames per second or Mpps (million packets per second)

Bridge Limitations

Broadcast Domain:

- All connected segments form single broadcast domain
- Broadcast traffic forwarded to all segments
- Can cause performance issues in large networks

Spanning Tree Limitations:

- Blocks redundant links (wastes bandwidth)
- Slow convergence (30-50 seconds)
- Single point of failure (root bridge)

Scalability:

- MAC table size limits
- Broadcast traffic increases with network size
- STP convergence time increases with network complexity

Modern Developments

Rapid Spanning Tree (RSTP):

- IEEE 802.1w
- Faster convergence (2-3 seconds)
- Backward compatible with STP

Multiple Spanning Tree (MSTP):

- IEEE 802.1s
- Multiple spanning trees for different VLANs
- Better load balancing

VLANs (Virtual LANs):

- Logical segmentation of broadcast domains
- IEEE 802.1Q standard
- Reduces broadcast traffic, improves security

Ethernet Switches:

- Evolution of bridges
- Higher port density
- Additional features (VLANs, QoS, management)

- Full-duplex operation

Bridge Configuration

Basic Configuration:

```
Bridge Priority: 32768 (default)
MAC Address: 00:1A:2B:3C:4D:5E
Hello Time: 2 seconds
Max Age: 20 seconds
Forward Delay: 15 seconds
```

Port Configuration:

```
Port 1: Cost 19 (100 Mbps), State Forwarding
Port 2: Cost 4 (1 Gbps), State Forwarding
Port 3: Cost 19 (100 Mbps), State Blocking
```

Monitoring:

- MAC address table utilization
- Port statistics (frames forwarded, filtered, flooded)
- STP topology changes
- Error counters

GATE Tips:

- Bridges operate at Data Link Layer (Layer 2)
- Learn MAC addresses automatically (transparent learning)
- Three forwarding decisions: forward, filter, flood
- Spanning Tree Protocol prevents loops in redundant topologies
- Root bridge elected based on lowest Bridge ID
- Port states: Blocking → Listening → Learning → Forwarding
- Bridges create separate collision domains but single broadcast domain
- Store-and-forward vs. cut-through forwarding trade-offs
- Modern switches are evolved bridges with additional features
- VLANs provide logical segmentation of broadcast domains

Missing Topics - Engineering Mathematics

LU Decomposition

Key Concepts: LU decomposition factors a matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U). Essential for solving systems of linear equations efficiently.

LU Decomposition Overview:

- **Purpose**: Factor matrix A into A = LU
- **L Matrix**: Lower triangular (elements above diagonal are 0)
- **U Matrix**: Upper triangular (elements below diagonal are 0)
- **Applications**: Solving linear systems, matrix inversion, determinant calculation
- **Advantage**: Solve multiple systems with same coefficient matrix efficiently

Mathematical Foundation:

Definition: For an n×n matrix A, LU decomposition finds matrices L and U such that:

$$A = LU$$

where:

- **L**: Lower triangular matrix with 1's on diagonal (unit lower triangular)
- **U**: Upper triangular matrix

Example:

$$\begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

Existence and Uniqueness:

- **Existence**: LU decomposition exists if all leading principal minors are non-zero
- **Uniqueness**: If L has unit diagonal, decomposition is unique
- **Alternative**: Can have U with unit diagonal instead of L

Gaussian Elimination Method

Algorithm: LU decomposition can be obtained through Gaussian elimination without pivoting.

Step-by-Step Process:

Step 1: Start with matrix A

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Step 2: Eliminate below first pivot

- Multiply row 1 by $m_{21} = a_{21}/a_{11}$ and subtract from row 2
- Multiply row 1 by $m_{31} = a_{31}/a_{11}$ and subtract from row 3

Step 3: Continue elimination for remaining columns

- Record multipliers in L matrix
- Resulting upper triangular matrix is U

Detailed Example:

Given Matrix:

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix}$$

Step 1: Eliminate column 1 below diagonal

- $m_{21} = 4/2 = 2$: Row 2 ← Row 2 - 2×Row 1
- $m_{31} = 8/2 = 4$: Row 3 ← Row 3 - 4×Row 1

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 3 & 5 \end{bmatrix}$$

Step 2: Eliminate column 2 below diagonal

- $m_{32} = 3/1 = 3$: Row 3 ← Row 3 - 3×Row 2

$$U = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

Step 3: Construct L matrix from multipliers

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{bmatrix}$$

Verification:

$$LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix} = A$$

✓

Doolittle's Method

Doolittle's Algorithm: Direct method to find L and U simultaneously.

Formulas:

For U matrix (row by row):

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \quad \text{for } j \geq i$$

For L matrix (column by column):

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \quad \text{for } i > j$$

Diagonal of L: $l_{ii} = 1$ for all i

Algorithm Steps:

1. Set $l_{ii} = 1$ for all i
2. For each row i and column j:
   - If i ≤ j: Calculate u_{ij}
   - If i > j: Calculate l_{ij}

Example Implementation:

```
void doolittleLU(double A[][n], double L[][n], double U[][n]) {
    // Initialize L as identity matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            L[i][j] = (i == j) ? 1.0 : 0.0;
            U[i][j] = 0.0;
        }
    }

    // Decomposition
    for (int i = 0; i < n; i++) {
        // Upper triangular matrix U
        for (int j = i; j < n; j++) {
            U[i][j] = A[i][j];
            for (int k = 0; k < i; k++) {
                U[i][j] -= L[i][k] * U[k][j];
            }
        }

        // Lower triangular matrix L
        for (int j = i + 1; j < n; j++) {
            L[j][i] = A[j][i];
            for (int k = 0; k < i; k++) {
                L[j][i] -= L[j][k] * U[k][i];
            }
            L[j][i] /= U[i][i];
        }
    }
}
```

Crout's Method

Crout's Algorithm: Alternative method where L has arbitrary diagonal and U has unit diagonal.

Difference from Doolittle:

- **Doolittle**: L has unit diagonal, U has arbitrary diagonal
- **Crout**: L has arbitrary diagonal, U has unit diagonal

Formulas for Crout's Method:

For L matrix:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \quad \text{for } i \geq j$$

For U matrix:

$$u_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \quad \text{for } i < j$$

Diagonal of U: $u_{ii} = 1$ for all i

Solving Linear Systems with LU Decomposition

Problem: Solve Ax = b where A = LU

Two-Step Process:

Step 1: Forward substitution to solve Ly = b

$$\begin{cases} l_{11} y_1 = b_1 \\ l_{21} y_1 + l_{22} y_2 = b_2 \\ l_{31} y_1 + l_{32} y_2 + l_{33} y_3 = b_3 \\ \vdots \end{cases}$$

Step 2: Backward substitution to solve Ux = y

$$\begin{cases} u_{nn} x_n = y_n \\ u_{n-1,n-1} x_{n-1} + u_{n-1,n} x_n = y_{n-1} \\ \vdots \end{cases}$$

Example: Solve Ax = b where A = LU from previous example and b = [5, 11, 21]ᵀ

Step 1: Solve Ly = b

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \\ 21 \end{bmatrix}$$

- $y_1 = 5$
- $2y_1 + y_2 = 11 \Rightarrow y_2 = 11 - 10 = 1$
- $4y_1 + 3y_2 + y_3 = 21 \Rightarrow y_3 = 21 - 20 - 3 = -2$

So $y = [5, 1, -2]^T$

Step 2: Solve Ux = y

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ -2 \end{bmatrix}$$

- $2x_3 = -2 \Rightarrow x_3 = -1$
- $x_2 + x_3 = 1 \Rightarrow x_2 = 1 - (-1) = 2$
- $2x_1 + x_2 + x_3 = 5 \Rightarrow x_1 = (5 - 2 - (-1))/2 = 2$

Solution: $x = [2, 2, -1]^T$

Partial Pivoting

Problem: LU decomposition fails if pivot element is zero or very small.

Solution: Partial pivoting - interchange rows to get largest element as pivot.

PLU Decomposition: PA = LU where P is permutation matrix.

Algorithm:

1. At each step, find row with largest absolute value in current column
2. Interchange rows if necessary
3. Record row interchanges in permutation matrix P
4. Continue with standard LU decomposition

Example with Pivoting:

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 3 \\ 1 & 0 & 1 \end{bmatrix}$$

Step 1: Pivot needed ($a_{11} = 0$)

- Largest element in column 1 is 2 (row 2)
- Interchange rows 1 and 2

$$P_1 A = \begin{bmatrix} 2 & 1 & 3 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Continue decomposition with pivoted matrix.

Applications of LU Decomposition

1. Solving Multiple Systems:

- Same coefficient matrix A, different right-hand sides b
- Decompose A once: A = LU
- For each b: solve Ly = b, then Ux = y
- **Complexity**: $O(n^3)$ for decomposition + $O(n^2)$ per system

2. Matrix Inversion:

- Solve AX = I where $X = A^{-1}$
- Equivalent to solving n systems: $Ax_i = e_i$
- Each e_i is i-th column of identity matrix

3. Determinant Calculation:

- det(A) = det(L) × det(U)
- det(L) = 1 (unit diagonal)
- det(U) = product of diagonal elements
- **Result**: $\det(A) = u_{11} \times u_{22} \times \ldots \times u_{nn}$

4. Matrix Condition Number:

- Estimate condition number using LU factors
- Important for numerical stability analysis

Computational Complexity

Time Complexity:

- **LU Decomposition**: $O(n^3/3)$ operations
- **Forward/Backward Substitution**: $O(n^2)$ each
- **Total for one system**: $O(n^3/3 + n^2) \approx O(n^3)$
- **Additional systems**: $O(n^2)$ each

Space Complexity:

- **In-place**: Can overwrite A with L and U
- **Storage**: $O(n^2)$ for n×n matrix
- **Additional**: O(n) for permutation vector (if pivoting)

Comparison with Other Methods:

Method	Time Complexity	Space	Stability	Use Case
Gaussian Elimination	$O(n^3)$	$O(n^2)$	Good with pivoting	Single system
LU Decomposition	$O(n^3) + O(n^2)$ per system	$O(n^2)$	Good with pivoting	Multiple systems
Iterative Methods	$O(kn^2)$	$O(n^2)$	Varies	Large sparse systems

GATE Tips:

- LU decomposition factors A = LU (lower × upper triangular)
- Doolittle: L has unit diagonal, U has arbitrary diagonal
- Crout: L has arbitrary diagonal, U has unit diagonal
- Solving Ax = b: First Ly = b, then Ux = y
- Partial pivoting needed for numerical stability
- Efficient for multiple systems with same coefficient matrix
- Determinant = product of U's diagonal elements
- Time complexity: $O(n^3)$ for decomposition, $O(n^2)$ per system
- Fails if leading principal minors are zero (need pivoting)
- Applications: linear systems, matrix inversion, determinant calculation