Step 1: Time and Space Complexity

- Big-O, Big-Ω, Big-Θ
- Amortized analysis
- Best / Average / Worst cases
- Tradeoffs

Step 2: Linear and Binary Search

Step 3: Array Patterns

1. SLIDING WINDOW

Used when the problem talks about **subarray / substring / continuous**.

A. Fixed Size Sliding Window

When to use

- Window size = **K**
- "Maximum / minimum / average of subarrays of size K"

Core Idea

- Expand right
- Shrink left when window size > K
- Maintain running sum/count

Template

```
initialize window
for r in range(n):
    add arr[r]
    if window size == k:
```

```
update answer
remove arr[l]
l++
```

LeetCode (MUST DO)

643. Maximum Average Subarray I

644. Number of Sub-arrays of Size K

645. Find All Anagrams in a String

646. Permutation in String

B. Variable Size Sliding Window

When to use

- "Longest / shortest subarray"
- Condition: sum ≥ k, at most k distinct, etc.

Core Idea

- Expand right
- Shrink left **while condition breaks**
- Update answer

LeetCode

209. Minimum Size Subarray Sum

210. Longest Substring Without Repeating Characters

211. Longest Repeating Character Replacement

212. Max Consecutive Ones III

213. Fruit Into Baskets

FAANG favorite

2. TWO POINTERS

Used when array is **sorted** or can be treated from **both ends**.

When to use

- Sorted array
- Pairs / triplets
- Palindrome check

Core Idea

- One pointer at start
- One at end
- Move based on condition

Template

```
l = 0, r = n-1
while l < r:
    if condition:
        l++
    else:
        r--
```

LeetCode

1. Two Sum (sorted version)
2. Two Sum II
3. 3Sum
4. 3Sum Closest
5. Container With Most Water
6. Valid Palindrome

3. PREFIX SUM

Used for **range queries** and **subarray sums**.

When to use

- “Sum from i to j”
- Multiple sum queries
- Count subarrays with given sum

Core Idea

```
prefix[i] = prefix[i-1] + arr[i]
sum(i,j) = prefix[j] - prefix[i-1]
```

LeetCode

303. Range Sum Query – Immutable

304. Subarray Sum Equals K

305. Contiguous Array

306. Find Pivot Index

307. Product of Array Except Self (prefix + suffix)

Combine with **HashMap** for maximum power

4. DIFFERENCE ARRAY

Used when **range updates** are many.

When to use

- "Apply Q range updates"
- Add value to range [l, r]

Core Idea

```
diff[l] += x
diff[r+1] -= x
final array = prefix sum of diff
```

LeetCode

370. Range Addition

371. Corporate Flight Bookings

372. Shifting Letters II

Often ignored, big advantage if you know this

5. KADANE'S ALGORITHM

Used for **maximum subarray sum**.

When to use

- “Maximum sum subarray”
- Continuous subarray

Core Idea

```
curr = max(arr[i], curr + arr[i])
maxSum = max(maxSum, curr)
```

LeetCode

53. Maximum Subarray
54. Maximum Sum Circular Subarray
55. Maximum Product Subarray

Amazon and Google staple

6. CYCLIC SORT

Used when numbers are in **range 1..N**.

When to use

- Missing number
- Duplicate
- Corrupt pair

Core Idea

Place each element at its correct index.

Template

```
while i < n:
    correct = arr[i] - 1
    if arr[i] != arr[correct]:
        swap
    else:
        i++
```

LeetCode

268. Missing Number
269. Find All Numbers Disappeared

270. Find All Duplicates

271. Find the Duplicate Number

7. DUTCH NATIONAL FLAG

Sorting arrays with **only 0s, 1s, 2s**.

When to use

- Three unique values
- In-place sorting

Core Idea

- low → 0
- mid → current
- high → 2

LeetCode

75. Sort Colors
76. Sort Array By Parity
77. Squares of a Sorted Array

8. MERGE INTERVALS

Used when dealing with **ranges**.

When to use

- Overlapping intervals
- Scheduling problems

Core Idea

1. Sort by start time
2. Merge if overlapping

LeetCode

56. Merge Intervals
57. Insert Interval
58. Non-overlapping Intervals
59. Interval List Intersections

Appears in system + DSA rounds

9. MATRIX TRAVERSAL

2D array problems.

A. Spiral Traversal

LeetCode

54. Spiral Matrix
55. Spiral Matrix II

B. Diagonal Traversal

LeetCode

498. Diagonal Traverse
499. Sort the Matrix Diagonally

C. General Matrix Patterns

- Boundary traversal
- Transpose
- Rotate matrix

LeetCode

48. Rotate Image
49. Set Matrix Zeroes
50. Search a 2D Matrix II

3. String patterns

Strings are **arrays with constraints**.
FAANG interviewers test whether you can **convert string problems into array + hashing + window patterns**.

1. FREQUENCY MAP

Used to **count characters** and compare occurrences.

When to use

- Counting characters
- Comparing two strings
- Checking constraints like “at most k”
- Anagrams, permutations, replacements

Core Idea

Use an array (size 26 / 128 / 256) or HashMap.

```
freq[c]++
freq[c]--
```

Choose:

- Array → lowercase letters, faster
- HashMap → general characters

Common Traps

- Forgetting case sensitivity
- Using HashMap when array suffices
- Not resetting counts properly

LeetCode (MUST DO)

242. Valid Anagram

243. First Unique Character in a String

244. Ransom Note

245. Longest Palindrome

246. Find Words That Can Be Formed by Characters

2. ANAGRAM PATTERNS

A direct extension of **frequency map + sliding window**.

When to use

- “Anagram”
- “Permutation”
- “Rearrange characters”

Core Idea

Two frequency arrays are equal → anagram.

For substrings:

- Maintain a sliding window
- Increment entering char
- Decrement leaving char

Typical Template (conceptual)

```
need = frequency of target
window = empty frequency
for r:
    add s[r]
    if window size > target size:
        remove s[l]
        l++
    if window == need:
        answer found
```

Common Traps

- Comparing maps every iteration (slow)
- Not shrinking window correctly
- Forgetting duplicate characters

LeetCode

242. Valid Anagram

243. Find All Anagrams in a String

244. Permutation in String

245. Group Anagrams

3. PALINDROME CHECKS

Palindrome problems test **two pointers + edge handling**.

When to use

- “Palindrome”
- “Same forward and backward”
- Ignore non-alphanumeric

Core Idea

Two pointers from both ends.

```
l = 0, r = n-1
while l < r:
    if invalid: move pointer
    if s[l] != s[r]: false
    l++, r--
```

Variations

- Case insensitive
- Ignore special characters
- At most one removal allowed

Common Traps

- Forgetting to skip non-alphanumeric
- Not handling empty strings
- Incorrect pointer movement

LeetCode

125. Valid Palindrome
126. Valid Palindrome II
127. Longest Palindromic Substring
128. Palindromic Substrings

4. SLIDING WINDOW ON STRINGS

This is **the most important string pattern for FAANG**.

When to use

- “Longest / shortest substring”
- Constraints like:
  - No repeating characters
  - At most k distinct
  - Replace at most k characters

Core Idea

- Expand right pointer
- Maintain frequency map
- Shrink left when condition breaks
- Track max/min length

Typical Conditions

- `freq[ch] > 1`

- `distinct > k`
- `window_size - maxFreq > k`

Common Traps

- Shrinking too early or too late
- Incorrect window condition
- Forgetting to update answer after shrink

LeetCode (VERY IMPORTANT)

3. Longest Substring Without Repeating Characters
4. Longest Repeating Character Replacement
5. Longest Substring with At Most K Distinct Characters
6. Max Consecutive Ones III (same pattern)
7. Minimum Window Substring

5. STRING MATCHING BASICS (KMP OVERVIEW)

FAANG usually tests **conceptual understanding**, not full KMP coding.

Problem with naive matching

- Worst case O(n × m)

KMP Core Idea

- Preprocess pattern
- Build **LPS array** (Longest Prefix which is also Suffix)
- Avoid rechecking characters

What interviewers expect

- Why KMP is faster
- What LPS represents

- How backtracking is avoided

LeetCode (Know at least one)

28. Find the Index of the First Occurrence in a String
29. Repeated Substring Pattern

Interview Tip

If KMP code is too long:

- Explain approach
- Mention time complexity
- Offer to code if required

6. ENCODING / DECODING

Tests **string building + delimiters + edge cases**.

When to use

- Encode list → single string
- Decode string → list
- Design problems

Core Idea

- Use length + delimiter
- Avoid ambiguity

Example concept:

```
encode: 4#leet5#code
decode: read length, extract substring
```

Common Traps

- Using delimiter without escaping
- Failing on empty strings

- Incorrect parsing logic

LeetCode

271. Encode and Decode Strings

272. Encode and Decode TinyURL

7. STRING HASHING (INTRO)

Used for **fast comparison of substrings**.

When to use

- Check substring equality
- Avoid direct string comparison
- Pattern matching

Core Idea

Convert string to numeric hash.

```
hash = (prev_hash * base + char) % mod
```

Use:

- Prefix hash
- Power array

What FAANG expects

- Conceptual understanding
- Collision awareness
- Why modulo is needed

LeetCode

187. Repeated DNA Sequences

188. Longest Duplicate Substring

189. Shortest Palindrome

Strings are not hard.
They are **arrays with rules**.

If you master:

- Frequency map
- Sliding window
- Two pointers

You automatically solve **70% of string problems**.

Step 4: Linked List patterns

1. FAST & SLOW POINTER (TORTOISE–HARE)

When to use

- Detect cycle
- Find middle node
- Find kth node from end
- Palindrome linked list

Core Idea

- `slow` moves 1 step
- `fast` moves 2 steps

If they meet → cycle exists.

Typical Patterns

- Middle of list → slow at middle
- Even length → clarify first or second middle
- Kth from end → move fast k steps first

Common Traps

- Not checking `fast != null && fast.next != null`
- Off-by-one errors in even length lists

LeetCode (MUST DO)

876. Middle of the Linked List

877. Linked List Cycle

878. Linked List Cycle II

879. Palindrome Linked List

2. REVERSE LINKED LIST

Reversal is the **foundation** of most hard LL problems.

A. Reverse Entire List

Core Idea (Iterative)

```
prev = null
curr = head
while curr:
    next = curr.next
    curr.next = prev
    prev = curr
    curr = next
return prev
```

Common Traps

- Losing reference to `next`
- Returning wrong head

LeetCode

206. Reverse Linked List

B. Reverse Partial List

When to use

- Reverse between positions `left` and `right`

Core Idea

- Traverse to `left-1`
- Reverse sublist
- Reconnect three parts

LeetCode

92. Reverse Linked List II

C. Reverse in K-Group

When to use

- Reverse nodes in groups of size `k`

Core Idea

- Check if k nodes exist
- Reverse k nodes
- Recursively or iteratively process rest

Common Traps

- Reversing last group when size < k
- Incorrect reconnection

LeetCode

25. Reverse Nodes in k-Group

3. MERGE LISTS

Tests **pointer coordination + sorted logic**.

When to use

- Two sorted linked lists
- Multiple sorted lists

Core Idea

- Compare nodes
- Attach smaller node
- Move pointer

Template

```
dummy = new Node(-1)
tail = dummy
while l1 && l2:
    if l1.val < l2.val:
        tail.next = l1
        l1 = l1.next
    else:
        tail.next = l2
        l2 = l2.next
    tail = tail.next
tail.next = l1 or l2
```


Common Traps

- Forgetting remaining nodes
- Not moving tail

LeetCode

21. Merge Two Sorted Lists
22. Merge k Sorted Lists

4. CYCLE DETECTION

Classic **fast & slow pointer application**.

A. Detect Cycle (Yes / No)

Core Idea

- If fast meets slow → cycle exists

LeetCode

141. Linked List Cycle

B. Find Cycle Starting Node

Core Idea

- After meeting point:
  - Move one pointer to head
  - Move both one step
  - Meeting point = cycle start

LeetCode

142. Linked List Cycle II

Common Traps

- Trying to use HashSet when O(1) is required
- Not understanding why pointers meet at cycle start

5. DUMMY NODE PATTERN

A **FAANG favorite for clean code**.

When to use

- Head may change

- Insertion / deletion at head
- Merging lists

Core Idea

Add a fake node before head to simplify logic.

Benefits

- Eliminates head special cases
- Cleaner pointer updates
- Less bugs

Common Use Cases

- Remove nth node from end
- Merge lists
- Partition list

LeetCode

19. Remove Nth Node From End of List
20. Remove Linked List Elements
21. Partition List

Step 5: Stack patterns

Stacks are used when the problem involves:

- **Previous / Next**
- **Undo / Backtracking**
- **Nested structures**
- **Last seen, first resolved**
- **Monotonic behavior**

1. BASIC STACK SIMULATION

When to use

- Simulate process step-by-step
- Matching symbols
- Undo / redo
- Backtracking

Core Idea

- Push when entering
- Pop when resolving

Common Traps

- Forgetting empty stack checks
- Incorrect pop order

LeetCode

20. Valid Parentheses
21. Remove All Adjacent Duplicates in String
22. Make The String Great
23. Backspace String Compare

2. MONOTONIC STACK (MOST IMPORTANT)

This is the **most frequently tested stack pattern in FAANG**.

What is a Monotonic Stack?

A stack that is:

- **Monotonically increasing** or

- **Monotonically decreasing**

When to use

- Next Greater / Smaller element
- Previous Greater / Smaller element
- Histogram / rectangle problems
- Temperature / stock span type problems

Core Idea

- Maintain order in stack
- Pop while condition breaks monotonicity

Generic Template

```
stack = empty
for i in range(n):
    while stack not empty and arr[stack.top] < arr[i]:
        stack.pop()
    stack.push(i)
```

(Change comparison based on problem)

Common Traps

- Forgetting to process remaining stack
- Using values instead of indices
- Wrong comparison sign

LeetCode (MUST DO)

496. Next Greater Element I
497. Next Greater Element II
498. Daily Temperatures
499. Online Stock Span
500. Largest Rectangle in Histogram

501. Sum of Subarray Minimums

3. NEXT / PREVIOUS GREATER OR SMALLER

This is a **direct application** of monotonic stack.

Variants

- Next Greater Element
- Next Smaller Element
- Previous Greater Element
- Previous Smaller Element

Core Idea

- Traverse left → right or right → left
- Stack holds unresolved elements

LeetCode

496. Next Greater Element I

497. Next Greater Node In Linked List

498. Final Prices With a Special Discount

4. STACK FOR RANGE / AREA PROBLEMS

When to use

- Largest rectangle
- Maximum area
- Range expansion problems

Core Idea

- For each element, find:

    - Nearest smaller to left
    - Nearest smaller to right
- Use width × height formula

LeetCode

84. Largest Rectangle in Histogram
85. Maximal Rectangle

Interview Insight

These problems test:

- Boundary handling
- Stack cleanup
- Mathematical reasoning

5. EXPRESSION EVALUATION

When to use

- Mathematical expressions
- Operator precedence
- Parentheses

Core Idea

- One stack for numbers
- One stack for operators
- Evaluate when precedence allows

LeetCode

150. Evaluate Reverse Polish Notation
151. Basic Calculator

152. Basic Calculator II

Common Traps

- Handling negative numbers
- Integer division rules
- Spaces in input

6. STACK + GREEDY

When to use

- Remove characters for best result
- Lexicographically smallest / largest
- Keep sequence minimal

Core Idea

- While stack top is worse than current
- And removals are allowed
- Pop stack

LeetCode

402. Remove K Digits

403. Remove Duplicate Letters

404. Smallest Subsequence of Distinct Characters

7. STACK FOR STRING DECODING

When to use

- Nested encoded strings
- Repetition with brackets

Core Idea

- Push current string and count
- Decode when `]` appears

LeetCode

394. Decode String
395. Number of Atoms

8. STACK FOR PARENTHESES VARIATIONS

When to use

- Longest valid parentheses
- Minimum removals
- Scoring parentheses

Core Idea

- Track indices or balance
- Use stack to mark valid ranges

LeetCode

32. Longest Valid Parentheses
33. Minimum Remove to Make Valid Parentheses
34. Score of Parentheses

9. STACK + RECURSION CONVERSION

When to use

- Convert recursion to iterative

- Avoid stack overflow
- Tree traversal simulation

Core Idea

- Manually manage call stack

LeetCode

144. Binary Tree Preorder Traversal
145. Binary Tree Inorder Traversal
146. Binary Tree Postorder Traversal

10. DESIGN PROBLEMS USING STACK

When to use

- Min / Max tracking
- Custom stack behavior

Core Idea

- Auxiliary stack for metadata

LeetCode

155. Min Stack
156. Max Stack
157. Implement Stack using Queues

If you master **Monotonic Stack + Stack Greedy**,
you cover **80% of FAANG stack questions**.

Do not memorize code.
Memorize **WHY elements are pushed and popped**.

Step 6: Queue and deque patterns

Queues are used when the problem requires:

- **First in, first out processing**
- **Level-by-level traversal**
- **Order preservation**
- **Window optimization**
- **Shortest / minimum steps**

1. BASIC QUEUE (FIFO SIMULATION)

When to use

- Process elements in arrival order
- Scheduling
- Streaming data

Core Idea

- Enqueue at back
- Dequeue from front

Common Traps

- Forgetting empty queue checks
- Incorrect order of processing

LeetCode

933. Number of Recent Calls

934. Number of Students Unable to Eat Lunch

935. Time Needed to Buy Tickets

2. BFS USING QUEUE (MOST IMPORTANT)

This is the **most common queue pattern in FAANG**.

When to use

- Shortest path (unweighted graph)
- Level-by-level processing
- Minimum steps / moves

Core Idea

- Push starting nodes
- Process level by level
- Mark visited to avoid repetition

Typical Template


```
queue.push(start)
while queue not empty:
    size = queue.size()
    for i in range(size):
        node = queue.pop()
        process node
        for each neighbor:
            if not visited:
                mark visited
                queue.push(neighbor)
```

Common Traps

- Forgetting to mark visited early
- Mixing BFS with DFS logic
- Incorrect level counting

LeetCode (MUST DO)

102. Binary Tree Level Order Traversal

103. Rotting Oranges

104. Word Ladder

105. Shortest Path in Binary Matrix

106. 01 Matrix

3. MULTI-SOURCE BFS

When to use

- Multiple starting points
- Spread / contamination problems
- Minimum distance from any source

Core Idea

- Push all sources initially
- BFS outward simultaneously

Common Traps

- Running BFS separately for each source
- Incorrect distance initialization

LeetCode

994. Rotting Oranges

995. 01 Matrix

996. As Far from Land as Possible

4. LEVEL ORDER WITH CONDITIONS

When to use

- Zigzag traversal
- Level-wise computations

- Grouping by depth

Core Idea

- Use queue size to separate levels
- Track level index

LeetCode

103. Binary Tree Zigzag Level Order Traversal

104. Find Largest Value in Each Tree Row

105. Average of Levels in Binary Tree

5. SLIDING WINDOW USING DEQUE (VERY IMPORTANT)

This is a **high-value FAANG pattern**.

When to use

- Maximum / minimum in window
- Continuous subarray optimization

Core Idea

- Deque stores indices
- Remove out-of-window indices
- Maintain decreasing or increasing order

Template

```
for i in range(n):
    remove indices out of window
    remove smaller elements from back
    add current index
    front of deque is answer
```

Common Traps

- Storing values instead of indices
- Not removing expired indices

LeetCode (MUST DO)

239. Sliding Window Maximum

240. Longest Continuous Subarray With Absolute Diff ≤ Limit

241. Shortest Subarray with Sum at Least K

6. CIRCULAR QUEUE

When to use

- Fixed size buffer
- Wrap-around indexing

Core Idea

- Use modulo arithmetic
- Track front, rear, size

Common Traps

- Confusing full vs empty
- Incorrect pointer movement

LeetCode

622. Design Circular Queue

623. Design Circular Deque

7. DEQUE FOR BIDIRECTIONAL PROCESSING

When to use

- Push/pop from both ends
- Reversal simulation
- Sliding window variants

Core Idea

- Use front and back operations wisely

LeetCode

1670.Design Front Middle Back Queue

1671.Number of Recent Calls (conceptually)

8. PRIORITY QUEUE (MIN / MAX HEAP) – QUEUE VARIANT

Used when **order depends on priority, not arrival**.

When to use

- Top-K problems
- Minimum / maximum extraction
- Scheduling with weights

Core Idea

- Heap ordered by priority
- Poll smallest / largest first

LeetCode

215. Kth Largest Element in an Array

216. Find Median from Data Stream

217. Kth Largest Element in a Stream

218. Top K Frequent Elements

9. QUEUE + GREEDY

When to use

- Optimal scheduling
- Reordering for best outcome

Core Idea

- Queue for order
- Greedy decision at each step

LeetCode

649. Dota2 Senate

650. Task Scheduler

Step 6: Binary Search Patterns

Binary Search is used when:

- Search space is **sorted** or **monotonic**
- Answer space can be **divided**
- Brute force is too slow

1. CLASSIC BINARY SEARCH

When to use

- Sorted array
- Exact element search

Core Idea

Divide search space into halves.

Template (Most Reliable)

```
l = 0, r = n - 1
while l <= r:
    mid = l + (r - l) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        l = mid + 1
    else:
        r = mid - 1
return -1
```

Common Traps

- Infinite loop (`l < r` vs `l <= r`)
- Overflow in mid calculation
- Missing return when not found

LeetCode

704. Binary Search

705. Guess Number Higher or Lower

2. FIRST / LAST OCCURRENCE

When to use

- Sorted array with duplicates
- Find boundary indices

Core Idea

Binary search with **biased movement**.

Template (First Occurrence)

```
while l <= r:
    mid = ...
    if arr[mid] >= target:
        r = mid - 1
    else:
        l = mid + 1
```

Template (Last Occurrence)

```
while l <= r:
    mid = ...
    if arr[mid] <= target:
        l = mid + 1
    else:
        r = mid - 1
```

LeetCode

34. Find First and Last Position of Element
35. First Bad Version

3. SEARCH IN ROTATED SORTED ARRAY

When to use

- Sorted array rotated at unknown pivot

Core Idea

- One half is always sorted
- Decide which half to discard

Common Traps

- Equal values
- Misidentifying sorted half

LeetCode

33. Search in Rotated Sorted Array
34. Search in Rotated Sorted Array II
35. Find Minimum in Rotated Sorted Array

4. PEAK ELEMENT / UNIMODAL SEARCH

When to use

- Increase then decrease
- Local maxima

Core Idea

- Compare mid with neighbors
- Move toward greater side

LeetCode

162. Find Peak Element
163. Peak Index in a Mountain Array

5. BINARY SEARCH ON ANSWER (MOST IMPORTANT)

This is the **highest-value binary search pattern in FAANG**.

When to use

- “Minimum / maximum value such that…”
- Answer range is numeric
- Condition is monotonic

Core Idea

- Search on **possible answers**

- Validate using a `check()` function

Template

```
l = min_possible
r = max_possible
while l < r:
    mid = l + (r - l) // 2
    if check(mid):
        r = mid
    else:
        l = mid + 1
return l
```

LeetCode (MUST DO)

875. Koko Eating Bananas

876. Capacity To Ship Packages Within D Days

877. Split Array Largest Sum

878. Minimum Number of Days to Make m Bouquets

879. Minimize Max Distance to Gas Station

6. BINARY SEARCH IN 2D MATRIX

When to use

- Matrix sorted row-wise / column-wise

Core Idea

- Treat matrix as 1D
- Or start from top-right corner

LeetCode

74. Search a 2D Matrix

75. Search a 2D Matrix II

BINARY SEARCH IN 2D MATRIX – ALL PATTERNS (FAANG COMPLETE)

PATTERN 1: FULLY SORTED MATRIX (ROW-MAJOR SORT)

Property

- Each row is sorted
- First element of a row > last element of previous row

Effectively, the matrix behaves like a **1D sorted array**.

When to use

- Global sorting guarantee
- "Search in a sorted matrix"

Core Idea

Flatten the matrix logically (not physically).

```
row = mid / cols
col = mid % cols
```

Time Complexity

- O(log (m × n))

LeetCode

74. Search a 2D Matrix

Common Traps

- Wrong row/column mapping
- Forgetting integer division

PATTERN 2: ROW-WISE AND COLUMN-WISE SORTED MATRIX (STAIRCASE SEARCH)

Property

- Rows sorted left → right
- Columns sorted top → bottom
- No global ordering across rows

When to use

- Matrix sorted both row-wise and column-wise
- Cannot flatten

Core Idea (Staircase Search)

Start from **top-right** or **bottom-left**.

```
if current > target: move left
if current < target: move down
```


Time Complexity

- O(m + n)

LeetCode

240. Search a 2D Matrix II

Why Binary Search Alone Fails Here

- Middle element gives no guarantee about all elements on one side

PATTERN 3: BINARY SEARCH ON EACH ROW (ROW-WISE ONLY SORTED)

Property

- Each row is sorted

- No relationship between rows

When to use

- Independent sorted rows
- Column order not guaranteed

Core Idea

- For each row:
    - Check range
    - Binary search inside row

Time Complexity

- O(m log n)

LeetCode

240. Search a 2D Matrix II (alternative approach)

241. Search a 2D Matrix (if condition weakened)

Common Traps

- Searching rows where target cannot exist
- Missing range pruning

PATTERN 4: BINARY SEARCH ON ANSWER IN 2D MATRIX (ADVANCED)

This is **often missed** and asked in **Google / Meta onsite rounds**.

Property

- Matrix values represent cost / distance / height
- Need to find minimum / maximum feasible value

When to use

- “Minimum value such that path exists”
- “Maximum threshold satisfying condition”

Core Idea

- Binary search on value
- Use BFS / DFS to validate

LeetCode

778. Swim in Rising Water

779. Path With Minimum Effort

780. Graph Valid Tree (conceptually similar)

Time Complexity

- O(log(max−min) × BFS)

PATTERN 5: MEDIAN / KTH ELEMENT IN SORTED MATRIX

Property

- Rows sorted
- Columns may or may not be sorted

When to use

- Find kth smallest
- Find median

Core Idea

- Binary search on value range

- Count elements ≤ mid

LeetCode

378. Kth Smallest Element in a Sorted Matrix

379. Kth Smallest Number in Multiplication Table

Key Insight

You are **not searching index**, you are **searching value**.

DECISION TABLE (VERY IMPORTANT)

Matrix Property	Pattern to Use
Fully sorted globally	Flattened binary search
Row & column sorted	Staircase search
Only rows sorted	Binary search per row
Path / effort problems	Binary search on answer
Median / kth element	Binary search on value

7. FIND MIN / MAX IN MONOTONIC FUNCTION

When to use

- Function switches from false → true
- Cost function optimization

Core Idea

- Lower bound binary search

LeetCode

367. Valid Perfect Square

368. Sqrt(x)

369. Find the Smallest Divisor Given a Threshold

8. FLOATING POINT BINARY SEARCH

When to use

- Precision-based problems
- Decimal answers

Core Idea

- Binary search with tolerance

LeetCode

774. Minimize Max Distance to Gas Station

Step 7: Tree Patterns

Trees test:

- Recursion clarity
- State passing
- Bottom-up vs top-down thinking
- Time/space trade-offs

1. DFS TRAVERSAL PATTERNS (FOUNDATION)

Types

- Preorder (Root → Left → Right)
- Inorder (Left → Root → Right)
- Postorder (Left → Right → Root)

When to use

- DFS = depth based problems
- Recursion natural fit

LeetCode

144. Binary Tree Preorder Traversal
145. Binary Tree Inorder Traversal
146. Binary Tree Postorder Traversal

2. BFS / LEVEL ORDER PATTERNS

When to use

- Level-wise processing
- Shortest depth
- Width-based problems

Core Idea

Use queue, process by level size.

Variants

- Normal level order

- Zigzag
- Reverse level order
- Level aggregation

LeetCode

102. Binary Tree Level Order Traversal
103. Zigzag Level Order Traversal
104. Minimum Depth of Binary Tree
105. Binary Tree Right Side View
106. Largest Value in Each Tree Row

3. HEIGHT / DEPTH BASED PATTERNS

These are **bottom-up DFS** problems.

When to use

- Height
- Depth
- Diameter
- Balance checking

Core Idea

Each node returns height to parent.

```
height = 1 + max(left, right)
```

LeetCode

104. Maximum Depth of Binary Tree
105. Balanced Binary Tree
106. Diameter of Binary Tree

Common Trap

Mixing **global answer** with **return value** incorrectly.

4. PATH-BASED PATTERNS (VERY IMPORTANT)

When to use

- Root-to-leaf paths
- Any-to-any paths
- Path sum problems

Variants

1. Root → Leaf
2. Root → Any
3. Any → Any (hardest)

Core Idea

- Pass path sum down
- Or return best contribution upward

LeetCode

112. Path Sum

113. Path Sum II

114. Path Sum III

115. Binary Tree Maximum Path Sum

FAANG Favorite

124. Maximum Path Sum
This is a **must-master**.

5. LCA (LOWEST COMMON ANCESTOR)

When to use

- Relationship between nodes
- Hierarchy problems

Core Idea (Binary Tree)

```
if root == p or q → return root
left = LCA(left)
right = LCA(right)
if both non-null → root is LCA
else return non-null
```

LeetCode

236. Lowest Common Ancestor of a Binary Tree

237. Lowest Common Ancestor of a BST

Interview Insight

BST LCA is **iterative + value based**
Binary Tree LCA is **recursive + structure based**

6. BST-SPECIFIC PATTERNS

BST problems exploit **sorted inorder traversal**.

When to use

- Range queries
- Kth smallest/largest
- Validation

Core Idea

Inorder traversal gives sorted order.

LeetCode

98. Validate Binary Search Tree

99. Kth Smallest Element in a BST
100. Range Sum of BST
101. Search in a BST

7. TREE TRANSFORMATION PATTERNS

When to use

- Modify tree structure
- Flatten, invert, mirror

Core Idea

Postorder traversal works best.

LeetCode

226. Invert Binary Tree
227. Flatten Binary Tree to Linked List
228. Merge Two Binary Trees

Common Trap

Losing child pointers before re-linking.

8. SERIALIZATION / DESERIALIZATION

When to use

- Tree to string
- Design problems

Core Idea

Use preorder + null markers.

LeetCode

297. Serialize and Deserialize Binary Tree
298. Serialize and Deserialize BST

Interview Tip

Explain logic first, code later.

9. TREE DP (ADVANCED FAANG PATTERN)

This separates **average candidates from top ones**.

When to use

- Each node has multiple states
- Decisions depend on children

Core Idea

Each node returns **multiple values**.

LeetCode

337. House Robber III
338. Binary Tree Cameras
339. Maximum Path Sum

Example

```
return [take_node, skip_node]
```

10. VIEW & ORDER PATTERNS

When to use

- Vertical / top / bottom view

- Column-based grouping

Core Idea

- BFS + column index
- HashMap + sorting

LeetCode

987. Vertical Order Traversal of a Binary Tree

988. Binary Tree Vertical Order Traversal

11. BUILD TREE PATTERNS

When to use

- Given traversals
- Construct tree

Core Idea

- One traversal gives root
- Other gives subtree boundaries

LeetCode

105. Construct Binary Tree from Preorder and Inorder

106. Construct Binary Tree from Inorder and Postorder

MASTER DECISION TABLE

Problem Type	Pattern
Level based	BFS
Height / balance	DFS bottom-up
Path problems	DFS + state

Relationship	LCA
Sorted property	BST logic
Modify structure	Postorder
Multiple states	Tree DP

OTHER TYPES OF TREES – COMPLETE INTERVIEW GUIDE

Think of this as **tree taxonomy + when/why they exist**.

1. N-ARY TREE

What it is

- Each node can have **more than two children**
- Used to represent **hierarchies**

When it appears

- File systems
- Organization charts
- Menu / UI trees
- Game trees

Key Differences from Binary Tree

- Children stored as a list
- Traversal logic is same, but loop over children

Traversal Pattern

```
DFS:
for child in children:
    dfs(child)

BFS:
queue-based level order
```

LeetCode

589. N-ary Tree Preorder Traversal

590. N-ary Tree Postorder Traversal

591. N-ary Tree Level Order Traversal

Interview Expectation

- Understand traversal
- No left/right assumption

2. BALANCED BINARY TREES (CONCEPTUAL)

Balanced trees ensure **O(log n)** operations.

A. AVL TREE

Properties

- Height difference between left & right ≤ 1
- Strictly balanced

Operations

- Rotations (LL, RR, LR, RL)

Interview Focus

- **Why rotations are needed**
- Not expected to code fully unless advanced role

B. RED-BLACK TREE

Properties

- Self-balancing via coloring
- Less strict than AVL
- Used in real systems

Real Usage

- Java `TreeMap`

- C++ `map` / `set`

Interview Focus

- Why Red-Black Tree is preferred over AVL
- Guarantees O(log n)

3. HEAP (PRIORITY QUEUE TREE)

What it is

- Complete binary tree
- Heap property (min or max)

When to use

- Top K problems
- Scheduling
- Streaming data

Key Insight

Heap is a **tree concept**, usually implemented via **array**.

Operations

- Insert → bubble up
- Delete → bubble down

LeetCode

215. Kth Largest Element in an Array

216. Find Median from Data Stream

217. Top K Frequent Elements

4. SEGMENT TREE (RANGE QUERY TREE)

What it is

- Binary tree for **range queries**

- Each node represents a segment

When to use

- Range sum / min / max
- Frequent updates + queries

Key Insight

- Divide array into segments
- Query/update in O(log n)

Interview Expectation

- Explain structure
- Write basic build/query if asked

5. FENWICK TREE (BINARY INDEXED TREE)

What it is

- Optimized structure for prefix sums

When to use

- Dynamic prefix sum
- Faster + simpler than segment tree

Key Insight

- Uses bit manipulation
- 1-indexed

Operations

- Update → O(log n)
- Query → O(log n)

6. TRIE (PREFIX TREE)

What it is

- Tree of characters
- Each path forms a word

When to use

- Prefix search
- Auto-complete
- Dictionary problems

Key Insight

- Each node has 26 / 52 / 128 children
- Space-time tradeoff

LeetCode

208. Implement Trie

209. Design Add and Search Words

210. Word Search II

7. B-TREE / B+ TREE (DATABASE TREES)

What it is

- Multi-way balanced tree
- Optimized for disk access

When it appears

- Databases
- File systems

Key Insight

- Fewer disk reads
- Nodes store multiple keys

Interview Expectation

- Conceptual understanding only

- Not coding

8. SUFFIX TREE / SUFFIX TRIE (ADVANCED STRING TREES)

What it is

- Tree of all suffixes of a string

When to use

- Substring queries
- Pattern matching
- Bioinformatics

Interview Expectation

- Know **what problem it solves**
- Not expected to implement

9. CARTESIAN TREE

What it is

- Heap + inorder traversal property

When to use

- Range minimum queries
- Convert array to tree

Insight

- Inorder traversal = original array
- Root = min/max element

10. DISJOINT SET TREE (UNION-FIND)

Technically not a tree structure you traverse, but **tree-based internally**.

When to use

- Connected components
- Cycle detection
- Kruskal’s algorithm

Optimizations

- Path compression
- Union by rank

QUICK INTERVIEW DECISION TABLE

Tree Type	Primary Use
Binary Tree	General hierarchy
BST	Ordered data
Heap	Priority / Top K
Trie	Prefix / words
Segment Tree	Range queries
Fenwick Tree	Prefix sums
Red-Black Tree	Balanced map/ set
B-Tree	Databases
N-ary Tree	Hierarchies
Suffix Tree	Substring search
Union-Find	Connectivity

Step 8: Hashing & Set Patterns

Hashing is used when you need:

- **Fast lookup (O(1))**
- **Frequency counting**
- **Uniqueness**
- **Prefix state tracking**
- **De-duplication**

1. BASIC HASH MAP LOOKUP

When to use

- Check existence
- Map keys to values
- Avoid nested loops

Core Idea

Store seen elements and query in O(1).

Common Traps

- Forgetting default values
- Overusing HashMap when array works
- Modifying map while iterating

LeetCode

1. Two Sum
2. Contains Duplicate
3. Contains Duplicate II
4. Intersection of Two Arrays

2. FREQUENCY COUNTING (CORE PATTERN)

When to use

- Count occurrences
- Anagrams
- Majority / mode problems

Core Idea

```
map[x] = map.get(x, 0) + 1
```

Common Traps

- Negative counts
- Forgetting to remove zero-count entries
- Wrong key type

LeetCode

242. Valid Anagram

243. Ransom Note

244. Majority Element

245. Sort Characters By Frequency

3. HASH SET FOR UNIQUENESS

When to use

- Remove duplicates
- Detect cycles
- Ensure uniqueness

Core Idea

```
if x in set → duplicate
else add to set
```

Common Traps

- Using list instead of set
- Forgetting memory trade-off

LeetCode

128. Longest Consecutive Sequence

129. Happy Number

130. Linked List Cycle (hashing approach)

4. PREFIX SUM + HASH MAP (VERY IMPORTANT)

When to use

- Subarray sum = K
- Zero sum subarrays
- Count of ranges

Core Idea

- Store prefix sum counts
- If `currSum - k` exists → valid subarray

Template (Conceptual)

```
map[0] = 1
sum = 0
for x in arr:
    sum += x
    if sum - k in map:
        count += map[sum - k]
    map[sum]++
```

Common Traps

- Forgetting `map[0] = 1`
- Updating map before checking
- Integer overflow

LeetCode (MUST DO)

560. Subarray Sum Equals K

561. Contiguous Array

562. Subarray Sums Divisible by K

5. HASHING + SLIDING WINDOW

When to use

- Longest substring problems
- At most K distinct
- Replacement constraints

Core Idea

- HashMap tracks window state
- Two pointers control window

LeetCode

3. Longest Substring Without Repeating Characters
4. Longest Repeating Character Replacement
5. Longest Substring with At Most K Distinct Characters

6. GROUPING BY HASH KEY

When to use

- Group anagrams
- Bucket data
- Normalize and group

Core Idea

- Transform element into canonical form
- Use as hash key

Common Keys

- Sorted string
- Frequency tuple
- Signature string

LeetCode

49. Group Anagrams
50. Sort Characters By Frequency
51. Top K Frequent Words

7. HASHING FOR CYCLE DETECTION

When to use

- Detect loops
- Repeated state detection

Core Idea

- Store visited states

Common Traps

- Memory overhead
- Missing termination condition

LeetCode

202. Happy Number
203. Linked List Cycle (hashing)
204. Circular Array Loop

8. HASHING FOR FAST LOOKBACK (STATE MEMORY)

When to use

- Remember last index
- Distance-based conditions
- Re-visiting logic

Core Idea

Store index or timestamp.

LeetCode

219. Contains Duplicate II
220. Bulls and Cows
221. Two Sum III – Data Structure Design

9. STRING HASHING (ROLLING HASH – INTRO)

When to use

- Compare substrings quickly
- Avoid repeated string comparisons
- Detect duplicates

Core Idea

- Convert string to number
- Use prefix hash

LeetCode

187. Repeated DNA Sequences
188. Longest Duplicate Substring
189. Longest Happy Prefix

10. SET + GREEDY PATTERNS

When to use

- Select unique optimal elements
- Dedup + greedy choice

Core Idea

- Maintain set of chosen items
- Greedy decision per step

LeetCode

316. Remove Duplicate Letters

317. Smallest Subsequence of Distinct Characters

11. HASH MAP DESIGN PROBLEMS

When to use

- Custom data structures
- Cache-like behavior

Core Idea

- HashMap + Doubly Linked List
- Or HashMap + Queue

LeetCode

146. LRU Cache

147. LFU Cache

Step 9: Recursion and backtracking

Recursion is about **problem decomposition**.
Backtracking is about **exploring choices and undoing them**.

1. RECURSION FUNDAMENTALS (NON-NEGOTIABLE)

When to use

- Tree traversal
- Divide problems into smaller identical problems
- When stack naturally represents state

Core Questions (Always Answer These First)

1. What is the **base case**?
2. What does the function **return**?
3. What state changes per call?
4. Does recursion go **deeper or narrower**?

Basic Template

```
func(params):
    if base case:
        return
    do work
    func(smaller params)
```

Common Traps

- Missing base case
- Infinite recursion
- Incorrect return propagation

LeetCode (Basics)

509. Fibonacci Number

510. Reverse String

511. Reverse Linked List (recursive)

2. TOP-DOWN VS BOTTOM-UP RECURSION

Top-Down

- Pass state down
- Result accumulated externally

Used when:

- Paths
- Constraints accumulate

Bottom-Up

- Children return result
- Parent computes answer

Used when:

- Height, depth, DP on trees

LeetCode

104. Maximum Depth of Binary Tree

105. Diameter of Binary Tree

3. MULTI-BRANCH RECURSION (DECISION TREE)

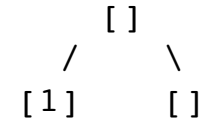
When to use

- Multiple choices at each step
- Explore all combinations

Core Idea

Each recursive call represents a **decision**.

Visual



LeetCode

78. Subsets
79. Permutations
80. Combinations

4. BACKTRACKING (UNDO AFTER RECURSION)

When to use

- Generate all valid answers
- Constraints must be respected

Core Idea

1. Choose
2. Explore
3. Undo

Template

```
backtrack(state):
    if solution:
        save state
        return
    for choice in choices:
        choose
        backtrack(state)
        undo
```

Common Traps

- Forgetting to undo
- Modifying shared state
- Missing pruning

5. SUBSETS PATTERN

When to use

- Power set
- Yes / No choice per element

Core Idea

At each index:

- Take element
- Skip element

LeetCode

78. Subsets
79. Subsets II

Interview Insight

This is the **simplest backtracking pattern**.

6. PERMUTATIONS PATTERN

When to use

- Order matters
- Rearrangements

Core Idea

- Fix one position
- Try all unused elements

LeetCode

46. Permutations
47. Permutations II

Common Traps

- Duplicate permutations
- Forgetting visited array

7. COMBINATIONS PATTERN

When to use

- Order doesn’t matter
- Choose k elements

Core Idea

- Use index-based recursion
- Avoid reusing previous elements

LeetCode

77. Combinations
78. Combination Sum
79. Combination Sum II

8. CONSTRAINT SATISFACTION PROBLEMS

When to use

- Board-based problems
- Rules to satisfy

Core Idea

- Try placing
- Check validity
- Backtrack on failure

LeetCode

51. N-Queens
52. N-Queens II
53. Sudoku Solver

9. STRING PARTITIONING BACKTRACKING

When to use

- Partition string
- Valid substring constraints

Core Idea

- Cut string at positions
- Validate each piece

LeetCode

131. Palindrome Partitioning
132. Restore IP Addresses

10. GRID / MATRIX BACKTRACKING

When to use

- Path finding
- Maze traversal
- DFS with backtracking

Core Idea

- Mark visited
- Explore neighbors
- Unmark on backtrack

LeetCode

79. Word Search
80. Number of Islands (DFS)
81. Max Area of Island

11. PRUNING (MOST IMPORTANT OPTIMIZATION)

When to use

- Search space explodes
- Constraints allow early stopping

Core Idea

- Stop exploring invalid branches early

Examples

- If sum > target → stop
- If length > limit → stop

LeetCode

39. Combination Sum
40. N-Queens

12. RECURSION TO DP TRANSITION (VERY IMPORTANT)

When to use

- Overlapping subproblems
- Exponential recursion

Core Idea

- Add memoization

LeetCode

70. Climbing Stairs
71. Word Break
72. Coin Change

Step 10: Heap patterns

Heaps answer one core question efficiently:

“What is the **minimum or maximum** element **right now**?”

1. BASIC HEAP OPERATIONS (FOUNDATION)

When to use

- Dynamic min / max retrieval
- Repeated extraction of extreme value

Core Idea

- Min-heap → smallest on top
- Max-heap → largest on top

Common Traps

- Forgetting heap size constraints
- Confusing min-heap vs max-heap
- Custom comparator mistakes

LeetCode

215. Kth Largest Element in an Array

216. Kth Largest Element in a Stream

2. TOP-K PATTERN (MOST IMPORTANT)

When to use

- Find K largest / smallest
- Stream of data
- Memory constraints

Core Idea

- Maintain heap of size K
- Pop when size > K

Which heap to use?

- K largest → **min-heap**
- K smallest → **max-heap**

LeetCode (MUST DO)

215. Kth Largest Element in an Array
216. Top K Frequent Elements
217. K Closest Points to Origin
218. Find K Closest Elements

Common Traps

- Using wrong heap type
- Sorting instead of heap
- Ignoring frequency map when needed

3. K-WAY MERGE PATTERN

When to use

- Merge multiple sorted lists / arrays
- Merge sorted streams

Core Idea

- Push first element of each list
- Pop smallest, push next from same list

Time Complexity

- O(N log K)

LeetCode

23. Merge k Sorted Lists
24. Find K Pairs with Smallest Sums
25. Kth Smallest Element in a Sorted Matrix

4. MEDIAN OF STREAM (TWO HEAPS PATTERN)

When to use

- Running median
- Streaming data

Core Idea

- Max-heap for left half
- Min-heap for right half
- Balance sizes

LeetCode

295. Find Median from Data Stream

Invariant

```
|left.size - right.size| ≤ 1
left.max ≤ right.min
```


5. FREQUENCY + HEAP

When to use

- Top K frequent elements
- Sorting by frequency

Core Idea

- Count frequency using HashMap
- Push into heap based on frequency

LeetCode

347. Top K Frequent Elements

348. Top K Frequent Words

349. Sort Characters By Frequency

6. SCHEDULING / INTERVAL PROBLEMS

When to use

- Rooms allocation
- CPU / task scheduling
- Overlapping intervals

Core Idea

- Sort by start time
- Min-heap tracks earliest ending resource

LeetCode

253. Meeting Rooms II

254. Task Scheduler

255. Reorganize String

7. HEAP + GREEDY

When to use

- Optimal choices at each step
- Maximize / minimize outcome

Core Idea

- Push candidates into heap
- Pick best available

LeetCode

502. IPO

503. Course Schedule III

504. Furthest Building You Can Reach

8. SLIDING WINDOW + HEAP

When to use

- Windowed min / max
- When deque solution is hard

Core Idea

- Heap stores (value, index)
- Lazy removal of expired elements

LeetCode

239. Sliding Window Maximum (heap approach)

240. Sliding Window Median

9. HEAP FOR GRAPH ALGORITHMS

When to use

- Shortest path
- Minimum spanning tree

Core Idea

- Greedy selection of smallest weight

LeetCode

743. Network Delay Time

744. Path with Maximum Probability

10. DESIGN PROBLEMS USING HEAP

When to use

- Custom priority logic
- Combined data structures

Core Idea

- Heap + HashMap / Set

LeetCode

355. Design Twitter

356. Find Median from Data Stream

357. LFU Cache

Step 11: Graph patterns

Graphs are used when problems involve:

- Relationships
- Dependencies
- Connectivity
- Paths / distances
- Ordering constraints

If you identify the **graph type and pattern correctly**, the problem becomes straightforward.

15. GRAPH BASICS (FOUNDATION)

Graph Representations

Adjacency List

- Most commonly used in interviews
- Space efficient for sparse graphs

```
graph[u] = [v1, v2, v3]
```

Time

- Traversal: O(V + E)

Adjacency Matrix

- Useful for dense graphs
- Quick edge lookup

```
matrix[u][v] = 1
```

Time

- Traversal: $O(V^2)$

Directed vs Undirected Graph

Feature	Directed	Undirected
Edge direction	One-way	Two-way
Cycle types	Directed cycles	Undirected cycles
In-degree / Out-degree	Yes	No
Use cases	Dependencies	Connectivity

Interview Rule

Always ask:

Is the graph directed or undirected?

This changes **cycle detection, traversal, and validity logic**.

1. GRAPH TRAVERSAL

Traversal is the **base layer** for all graph problems.

A. BFS (Breadth First Search)

When to use

- Shortest path (unweighted)
- Level-wise processing
- Minimum steps

Core Idea

Queue + visited set.

Template (Conceptual)

```
queue.push(start)
mark visited
while queue not empty:
    node = queue.pop()
    for neighbor:
        if not visited:
            mark visited
            queue.push(neighbor)
```

LeetCode

102. Binary Tree Level Order Traversal

103. Word Ladder

104. Rotting Oranges

B. DFS (Depth First Search)

When to use

- Exhaustive traversal
- Path existence
- Component exploration

Core Idea

Recursive or stack-based exploration.

LeetCode

200. Number of Islands

201. Max Area of Island

202. Number of Provinces

2. CONNECTED COMPONENTS

When to use

- Count isolated groups
- Check connectivity

Core Idea

Run DFS/BFS from every unvisited node.

LeetCode

200. Number of Islands

201. Number of Provinces

202. Number of Connected Components in an Undirected Graph

3. CYCLE DETECTION

Undirected Graph

Core Idea

- DFS with parent tracking
- If visited neighbor ≠ parent → cycle

LeetCode

261. Graph Valid Tree

Directed Graph

Core Idea

- DFS with recursion stack
- Back-edge → cycle

LeetCode

207. Course Schedule

208. Find Eventual Safe States

4. BIPARTITE CHECK

When to use

- Graph coloring
- Conflict detection
- Even-length cycle detection

Core Idea

- Two-color graph using BFS/DFS
- Adjacent nodes must have opposite colors

LeetCode

785. Is Graph Bipartite?

786. Possible Bipartition

5. SHORTEST PATH PATTERNS

A. DIJKSTRA'S ALGORITHM

When to use

- Non-negative weights

- Single source shortest path

Core Idea

- Greedy
- Min-heap to pick smallest distance

Complexity

- O((V + E) log V)

LeetCode

743. Network Delay Time

744. Path with Maximum Probability

Important Rule

Dijkstra FAILS with negative weights.

B. BELLMAN–FORD

When to use

- Negative weights allowed
- Need to detect negative cycle

Core Idea

- Relax all edges V−1 times
- One extra iteration to detect cycle

Complexity

- O(V × E)

LeetCode

- Negative Cycle Detection problems (theory heavy)

C. FLOYD–WARSHALL

When to use

- All-pairs shortest path
- Small graph

Core Idea

- DP on intermediate nodes

Complexity

- $O(V^3)$

LeetCode

1334.Find the City With the Smallest Number of Neighbors

D. 0–1 BFS

When to use

- Edge weights are only 0 or 1

Core Idea

- Deque instead of priority queue
- Push front for 0-cost edges
- Push back for 1-cost edges

Complexity

- $O(V + E)$

LeetCode

1368.Minimum Cost to Make at Least One Valid Path

1369.Minimum Obstacle Removal to Reach Corner

6. ADVANCED GRAPH PATTERNS

A. TOPOLOGICAL SORT

When to use

- Dependency resolution
- DAG ordering

Core Idea

- Kahn's algorithm (BFS + indegree)
- Or DFS + stack

Valid only for DAG

LeetCode

207. Course Schedule

208. Course Schedule II

7. UNION FIND (DISJOINT SET UNION – DSU)

When to use

- Connectivity queries
- Cycle detection
- Group merging

Core Idea

- Parent array
- Path compression
- Union by rank

LeetCode

684. Redundant Connection

685. Number of Provinces

686. Number of Operations to Make Network Connected

8. MINIMUM SPANNING TREE (MST)

When to use

- Minimum cost to connect all nodes

KRUSKAL'S ALGORITHM

- Sort edges
- Add if no cycle (DSU)

PRIM'S ALGORITHM

- Start from node
- Expand using min-heap

LeetCode

1584.Min Cost to Connect All Points

1585.Connecting Cities With Minimum Cost

9. BRIDGES & ARTICULATION POINTS (ADVANCED)

When to use

- Critical connections
- Network vulnerability

Core Idea

- DFS with discovery time
- Low-link values

LeetCode

1192.Critical Connections in a Network

Interview Expectation

- Conceptual understanding
- Not full implementation unless senior role

MASTER DECISION TABLE (GRAPHS)

Problem Type	Pattern
Connectivity	DFS / BFS
Shortest path (no weight)	BFS
Shortest path (weights)	Dijkstra
Negative edges	Bellman-Ford
Dependencies	Topological Sort
Cycle detection	DFS / DSU
Grouping	Union Find
Minimum connection cost	MST
Critical edges	Tarjan

Step 12: DP patterns

Dynamic Programming problems fall into **repeatable pattern buckets**.
If you identify the bucket correctly, the solution becomes mechanical.

1. LINEAR DP (1D DP)

When to use

- Problem progresses index by index
- State depends on previous index(es)

State

```
dp[i] = best answer up to index i
```

Transitions

- From `i-1`, `i-2`, etc.

LeetCode

70. Climbing Stairs
71. House Robber
72. Min Cost Climbing Stairs

2. 0/1 KNAPSACK PATTERN

When to use

- Each item can be chosen **once**
- Choose / skip decisions

State

```
dp[i][w] = best using first i items with capacity w
```

LeetCode

416. Partition Equal Subset Sum
417. Target Sum

418. Last Stone Weight II

3. UNBOUNDED KNAPSACK PATTERN

When to use

- Items can be reused unlimited times

Key Difference

After choosing an item, **stay at same index**.

LeetCode

322. Coin Change

323. Coin Change II

324. Word Break

4. SUBSET / BOOLEAN DP

When to use

- Check feasibility (true/false)
- Subset existence

State

```
dp[i][sum] = can we form sum
```

LeetCode

416. Partition Equal Subset Sum

417. Partition to K Equal Sum Subsets

5. SEQUENCE DP (LIS / LCS FAMILY)

When to use

- Order matters
- Non-contiguous subsequences

LIS State

```
dp[i] = LIS ending at i
```

LCS State

```
dp[i][j] = LCS up to i, j
```

LeetCode

300. Longest Increasing Subsequence

301. Longest Common Subsequence

302. Delete Operation for Two Strings

6. STRING DP

When to use

- String transformations
- Matching / editing

Common Problems

- Edit distance
- Palindromes
- Deletions / insertions

LeetCode

72. Edit Distance

73. Palindromic Substrings

74. Longest Palindromic Subsequence

7. MATRIX / GRID DP

When to use

- 2D grid
- Moves in directions

State

```
dp[i][j] = best way to reach (i, j)
```

LeetCode

62. Unique Paths
63. Minimum Path Sum
64. Maximal Square

8. INTERVAL / PARTITION DP

When to use

- Split array/string into parts
- Cost depends on partition

State

```
dp[i][j] = best answer for interval i to j
```

LeetCode

312. Burst Balloons
313. Minimum Score Triangulation
314. Minimum Cost to Cut a Stick

9. TREE DP

When to use

- Choices at tree nodes
- Parent-child dependency

State

Return multiple values from each node.

LeetCode

337. House Robber III
338. Binary Tree Cameras
339. Binary Tree Maximum Path Sum

10. GRAPH DP (DAG DP)

When to use

- Directed Acyclic Graph
- Dependencies between states

Core Idea

- Topological order + DP

LeetCode

329. Longest Increasing Path in a Matrix
330. Sort Items by Groups Respecting Dependencies

11. COUNTING DP

When to use

- Count number of ways
- Large answer, modulo needed

LeetCode

91. Decode Ways
92. Coin Change II
93. Distinct Subsequences

12. STATE COMPRESSION DP (BITMASK DP)

When to use

- Small N (≤ 15–20)
- Track visited states

State

```
dp[mask][i]
```

LeetCode

847. Shortest Path Visiting All Nodes

848. Partition to K Equal Sum Subsets

13. DIGIT DP (ADVANCED)

When to use

- Count numbers in a range
- Digit constraints

State

```
dp[pos][tight][state]
```

LeetCode

233. Number of Digit One

234. Numbers With Repeated Digits

14. DP WITH GREEDY OPTIMIZATION

When to use

- Greedy choice seems right
- DP proves optimality

LeetCode

45. Jump Game II

46. Palindrome Partitioning II

15. DP OPTIMIZATION PATTERNS

Techniques

- Space optimization (rolling array)
- Prefix/suffix optimization
- Monotonic queue optimization
- Binary search + DP

LeetCode

300. LIS (binary search optimization)

301. Sliding Window Maximum (DP + deque)

MASTER DP DECISION TABLE

Problem Clue	Pattern
Choose or skip	0/1 Knapsack
Reuse items	Unbounded Knapsack
Count ways	Counting DP
Subsequence	LIS / LCS
Grid movement	Matrix DP
Split problem	Interval DP
Tree choices	Tree DP
Small N + visited	Bitmask DP
Range counting	Digit DP

Step 13: Bit Manipulation

Bit manipulation is used when you need:

- Constant space
- Fast operations
- Compact state representation
- XOR-based reasoning

1. BASIC BIT OPERATIONS (FOUNDATION)

Core Operations

Operation	Meaning
`x & 1`	Check if odd
`x << 1`	Multiply by 2
`x >> 1`	Divide by 2
`x & (x - 1)`	Remove lowest set bit
`x ^ x`	0

Common Traps

- Forgetting signed vs unsigned shift
- Integer overflow
- Bit-width assumptions

LeetCode

191. Number of 1 Bits

192. Hamming Distance

2. CHECK / SET / CLEAR / TOGGLE BIT

When to use

- Flags
- State tracking
- Bitmask DP

Core Patterns

```
Check:   (x >> i) & 1
Set:     x | (1 << i)
Clear:   x & ~(1 << i)
Toggle:  x ^ (1 << i)
```

LeetCode

190. Reverse Bits

191. Bitwise AND of Numbers Range

3. XOR PATTERNS (MOST IMPORTANT)

When to use

- Find unique element
- Cancel pairs
- Parity logic

Core Properties

- `x ^ x = 0`
- `x ^ 0 = x`
- XOR is commutative

LeetCode

136. Single Number

137. Missing Number

138. Find the Difference

4. TWO UNIQUE NUMBERS USING XOR

When to use

- Exactly two unique elements
- Others appear twice

Core Idea

- XOR all → get `a ^ b`
- Extract rightmost set bit

- Partition array

LeetCode

260. Single Number III

5. BIT COUNTING / POPCOUNT

When to use

- Count set bits
- Parity
- Mask evaluation

Core Trick

```
while x:
    x &= (x - 1)
    count++
```


LeetCode

191. Number of 1 Bits

192. Counting Bits

6. POWER OF TWO PATTERN

When to use

- Check single bit set

Core Check

```
x > 0 and (x & (x - 1)) == 0
```

LeetCode

231. Power of Two

232. Power of Four

7. BITMASKING FOR SUBSETS

When to use

- Generate subsets
- Small N (≤ 20)

Core Idea

Each number from `0` to `2^n - 1` represents a subset.

LeetCode

78. Subsets
79. Subsets II

8. STATE COMPRESSION USING BITS

When to use

- Visited tracking
- Memory optimization

Core Idea

- Each bit represents a state

LeetCode

847. Shortest Path Visiting All Nodes
848. Smallest Sufficient Team

9. BITWISE GREEDY

When to use

- Maximize or minimize XOR
- Bit-by-bit construction

Core Idea

- Start from MSB to LSB
- Decide greedily

LeetCode

421. Maximum XOR of Two Numbers in an Array

422. Flip Game

10. TRIE + BIT MANIPULATION

When to use

- Fast XOR queries

Core Idea

- Each bit is a trie level
- Prefer opposite bit

LeetCode

421. Maximum XOR of Two Numbers in an Array

11. BIT MANIPULATION IN DP

When to use

- Bitmask DP
- Traveling salesman style problems

Core Idea

`dp[mask][i]`

LeetCode

847. Shortest Path Visiting All Nodes

848. Can I Win

12. BITWISE RANGE PATTERNS

When to use

- AND / OR over range
- Observing prefix stability

Core Idea

- Shift until numbers equal

LeetCode

201. Bitwise AND of Numbers Range

Step 13: Number Theory

Number theory problems usually hide behind:

- Constraints that break brute force
- Divisibility logic
- Modular arithmetic
- Prime-related optimizations

1. MODULAR ARITHMETIC (FOUNDATION)

When to use

- Large numbers
- “Return answer modulo 1e9+7”
- Repeated additions/multiplications

Core Rules

```
(a + b) % m = ((a % m) + (b % m)) % m
(a * b) % m = ((a % m) * (b % m)) % m
(a - b + m) % m = (a - b) % m
```

Common Traps

- Overflow before modulo
- Negative modulo
- Forgetting modulo during multiplication

LeetCode

70. Climbing Stairs (mod variant)
71. Unique Paths (mod variant)
72. Count Vowels Permutation

2. PRIME NUMBER PATTERNS

A. PRIME CHECK

When to use

- Single number primality
- Constraints ≤ 10^9

Core Idea

Check till √n.

LeetCode

204. Count Primes

205. Prime Number of Set Bits

B. SIEVE OF ERATOSTHENES

When to use

- Many prime queries
- Range-based prime computation

Core Idea

Mark multiples as non-prime.

Time

- O(n log log n)

LeetCode

204. Count Primes

3. GCD / LCM PATTERNS

When to use

- Fractions
- Ratios
- Divisibility constraints

Core Idea

Euclidean Algorithm.

```
gcd(a, b) = gcd(b, a % b)
```

Properties

```
lcm(a, b) = (a * b) / gcd(a, b)
```

LeetCode

1071.Greatest Common Divisor of Strings

1072.Max Points on a Line

1073.X of a Kind in a Deck of Cards

4. DIVISOR ENUMERATION

When to use

- Factor counting
- Perfect numbers
- Divisor sums

Core Idea

Divisors come in pairs.

Complexity

- O(√n)

LeetCode

507. Perfect Number

508. Number of Common Factors

5. FAST EXPONENTIATION (BINARY EXPONENTIATION)

When to use

- Large powers
- Modular exponentiation

Core Idea

```
pow(a, n):
    if n == 0 → 1
    if n even → pow(a*a, n/2)
    else → a * pow(a, n-1)
```

Time

- O(log n)

LeetCode

50. Pow(x, n)
51. Super Pow

6. MODULAR INVERSE (ADVANCED)

When to use

- Division under modulo
- Combinations / probabilities

Core Idea (Prime Mod)

`a⁻¹ ≡ a^(m−2) mod m`
(Fermat's Little Theorem)

Interview Expectation

- Conceptual understanding
- Not heavy proofs

7. COMBINATORICS (NCR PATTERN)

When to use

- Counting ways

- Arrangements
- DP + math hybrid

Core Idea

```
nCr = n! / (r! * (n-r)!)
```
Under modulo → use modular inverse.

LeetCode

62. Unique Paths
63. Count All Valid Pickup and Delivery Options

8. COUNTING MULTIPLES / INCLUSION–EXCLUSION

When to use

- Count numbers divisible by something
- Overlapping conditions

Core Idea

```
count(a or b) = count(a) + count(b) - count(a and b)
```

LeetCode

1201.Ugly Number III

1202.Nth Magical Number

9. DIGIT-BASED NUMBER THEORY

When to use

- Sum of digits
- Digit constraints
- Range counting

Core Idea

- Process number digit by digit
- Often transitions into **Digit DP**

LeetCode

233. Number of Digit One

234. Sequential Digits

10. BIT + NUMBER THEORY HYBRID

When to use

- Powers of two
- Binary properties
- XOR math

Core Ideas

```
n & (n - 1) == 0 → power of two
```

LeetCode

231. Power of Two

232. Power of Four

233. Reverse Bits

11. MATHEMATICAL OBSERVATION PATTERN

When to use

- Constraints too large
- Brute force impossible

Core Idea

Look for:

- Periodicity
- Symmetry
- Formula simplification

LeetCode

319. Bulb Switcher
320. Nim Game
321. Minimum Moves to Equal Array Elements

MASTER DECISION TABLE (NUMBER THEORY)

Problem Clue	Pattern
Large numbers	Modulo arithmetic
Divisibility	GCD / LCM
Prime related	Sieve / √n check
Power computation	Binary exponentiation
Counting ways	Combinatorics
Range divisible	Inclusion–Exclusion
Digit constraints	Digit DP
Power of two	Bit tricks

Step 15: Tries & Strings

A **Trie (Prefix Tree)** is used when problems involve:

- Prefix-based queries
- Fast string lookup
- Hierarchical character decisions
- Bitwise decisions (XOR tries)

1. TRIE FUNDAMENTALS

What is a Trie

- Each node represents a character
- Root represents empty string
- Path from root → node forms a prefix

When to use

- Search many strings
- Prefix matching
- Replace repeated string comparisons

Core Operations

- Insert
- Search
- StartsWith

Time Complexity

- Insert/Search: O(L), L = word length
- Space: High (trade-off for speed)

LeetCode

208. Implement Trie (Prefix Tree)

2. PREFIX SEARCH PATTERN (MOST COMMON)

When to use

- “Starts with…”

- Dictionary lookup
- Prefix validation

Core Idea

- Traverse trie character by character
- If path exists → prefix exists

Common Traps

- Forgetting end-of-word marker
- Case sensitivity
- Using HashSet instead of Trie for prefix queries

LeetCode

208. Implement Trie

209. Design Add and Search Words Data Structure

3. AUTO-COMPLETE PATTERN

When to use

- Suggest words
- Predictive typing
- Ranking suggestions

Core Idea

1. Traverse trie to prefix node
2. DFS/BFS to collect all words below

Enhancements

- Store frequency at nodes

- Limit results (top K)
- Use heap for ranking

Interview Expectation

- Explain approach clearly
- Partial implementation is acceptable

LeetCode

1268.Search Suggestions System

4. XOR MAXIMUM (BITWISE TRIE)

This is a **very high-value FAANG pattern**.

When to use

- Maximize XOR
- Bitwise greedy decisions

Core Idea

- Build trie on binary representation
- At each bit, choose opposite bit if possible

Why Trie Works Here

Each bit decision is independent and greedy.

LeetCode

421. Maximum XOR of Two Numbers in an Array

422. Maximum XOR With an Element From Array

Common Traps

- Incorrect bit length

- Forgetting leading zeros
- Using normal trie instead of bit trie

5. WORD SEARCH (TRIE + BACKTRACKING)

When to use

- Find multiple words in grid
- Repeated prefix checking

Core Idea

- Insert all words into trie
- DFS on grid
- Prune when prefix doesn't exist

Why Trie is Required

Without Trie:

- Repeated searches
- Exponential time

With Trie:

- Early pruning

LeetCode

212. Word Search II

Common Traps

- Not marking visited cells
- Forgetting to unmark (backtracking)
- Duplicate results

6. ADVANCED TRIE VARIATIONS (AWARENESS)

A. COMPRESSED TRIE (RADIX TREE)

- Merges single-child chains
- Reduces space

B. TERNARY SEARCH TREE

- Memory-efficient alternative
- Character comparisons

C. SUFFIX TRIE / TREE

- Stores suffixes
- Used for substring problems

Interview Expectation

- Conceptual understanding only

7. TRIE VS HASHING (IMPORTANT COMPARISON)

Use Case	Trie	Hashing
Prefix search	Excellent	Poor
Exact lookup	Good	Excellent
Memory	High	Low
Auto-complete	Yes	No
XOR problems	Yes	No

Step 16: Advanced system & system thinking

These topics appear in:

- Senior SDE interviews
- FAANG onsite depth rounds
- LLD / hybrid DSA + design rounds

1. SEGMENT TREE

What it solves

- Fast **range queries** with **updates**
- Range sum / min / max / gcd

When to use

- Multiple queries + updates
- Static array is not enough
- Constraints up to 10^5 or more

Core Idea

- Binary tree over array segments
- Each node stores result for a range

```
root → [0..n-1]
left → [0..mid]
right → [mid+1..n-1]
```

Operations

Operation	Time
Build	O(n)
Query	O(log n)
Update	O(log n)

Common Variants

- Lazy propagation (range update)
- Segment tree for min/max/sum

Interview Expectation

- Explain structure
- Write basic build/query/update
- Lazy propagation → conceptual unless senior role

LeetCode

307. Range Sum Query – Mutable

308. The Skyline Problem (conceptual)

2. FENWICK TREE (BINARY INDEXED TREE)

What it solves

- **Prefix sums with updates**
- Lighter alternative to segment tree

When to use

- Prefix queries only
- No need for full range logic

Core Idea

- Tree stored in array
- Uses **bit manipulation**

```
i += (i & -i)
i -= (i & -i)
```

Operations

Operation	Time
Update	O(log n)
Query	O(log n)

Differences vs Segment Tree

Fenwick	Segment Tree
Simple	Complex
Prefix only	Any range
Low memory	Higher memory

Interview Expectation

- Know BIT logic
- Prefix sum problems

3. SPARSE TABLE (STATIC RANGE QUERY)

What it solves

- **Static range queries**
- RMQ (min/max) without updates

When to use

- No updates
- Many queries
- Idempotent operations (min/max/gcd)

Core Idea

Precompute results for powers of two.

```
dp[i][j] = result of range [i, i + 2^j - 1]
```

Complexity

Build	O(n log n)
Query	O(1)

Interview Expectation

- Conceptual understanding
- Rare coding unless competitive role

4. MO'S ALGORITHM (QUERY REORDERING)

What it solves

- Offline range queries
- When updates are expensive

When to use

- Many queries on static array
- Query cost high, update cheap

Core Idea

- Sort queries by block
- Move window incrementally

Complexity

- $O((n + q)\sqrt{n})$

Interview Expectation

- Recognition only
- Explanation of idea

Typical Use

- Count distinct elements in range
- Frequency-based queries

5. ROLLING HASH (STRING SYSTEM TOOL)

What it solves

- Fast substring comparison
- Duplicate detection

When to use

- Avoid $O(n^2)$ string comparisons
- String matching

Core Idea

```
hash[i] = (hash[i-1] * base + s[i]) % mod
```


Applications

- Longest duplicate substring
- Pattern matching
- Palindrome checks

Interview Expectation

- Explain collisions
- Mention double hashing
- Code prefix hash if asked

LeetCode

1044.Longest Duplicate Substring

1045.Repeated DNA Sequences

6. LRU CACHE (DSA + DESIGN HYBRID)

What it solves

- Cache eviction policy
- O(1) get / put

Core Idea

- HashMap + Doubly Linked List

Operations

Operation	Time
Get	O(1)
Put	O(1)

Why DLL is needed

- Maintain usage order
- Remove least recently used

Interview Expectation

- Clean design
- Correct pointer handling
- Thread safety discussion (senior)

LeetCode

146. LRU Cache

7. DESIGN PROBLEMS (LLD BASICS)

What is tested

- Class design

- Responsibility separation
- Extensibility

Common LLD Patterns

- Cache design
- Rate limiter
- Parking lot
- Logger
- Elevator

Core Principles

- SOLID
- Separation of concerns
- Interface-driven design

Example Components (LRU)

```
Cache
Node
DoublyLinkedList
EvictionPolicy
```

Interview Expectation

- Explain before coding
- Handle edge cases
- Discuss trade-offs

MASTER DECISION TABLE (ADVANCED TOPICS)

Requirement	Use
Range query + update	Segment Tree
Prefix sum	Fenwick Tree

Static RMQ	Sparse Table
Offline queries	Mo’s Algorithm
Fast substring check	Rolling Hash
Cache eviction	LRU
System logic	LLD