Leetcode Notes

1. Arrays

Pattern: Linear data, indexed, often fixed-length, often sorted/unsorted.

Strategies:

- Brute-force double loops: O(N^2), only when N is small or constrained.
- Sorting + two-pointers / binary search / sliding window.
- Use prefix sums for range queries.
- In-place modifications to achieve O(1) extra space.

Watch for: boundaries, duplicates, in-place vs. extra space, sorting cost.

Archetypes:

- Two-sum (sorted vs. unsorted)
- 3-sum / k-sum
- Remove duplicates in-place
- Rotate array / reverse sections
- Partitioning (Dutch National Flag)

Recognition cues: ask for pair sums, sliding window sum target, in-place deletion or rearrangement.

Technique Overview:
Arrays problems often need **iteration, prefix sums, two-pointers, or sorting** to reduce brute force from O(N^2) to O(N) or O(N log N).

Pseudocode Template:

```
initialize result
for i from 0 to n-1:
    update result based on current element
return result
```

---

2. HashMaps & HashSets

Pattern: Fast lookup (O(1) average), counts, grouping, complement search.

Strategies:

- Count frequencies.
- Use map/set for complements (e.g., two-sum no sorting).
- Use map of list for grouping or adjacency (e.g., grouping anagrams).
- Use set for “seen” detection or ensuring uniqueness.
  Watch for: hash collision edge cases, large memory usage, key types.

Archetypes:

- Two-sum (hash version)
- Subarray sum equals k
- Longest substring without repeating characters
- Top k frequent elements (use map then heap)
- Anagram grouping

Recognition cues: ask about frequencies, pairs that sum to target, longest substring without repetition, grouping similar.

Technique Overview:
Use hash structures for **fast lookups** (O(1) average). Perfect for problems involving **frequency counts, complements, or grouping**.

Pseudocode Template:

```
initialize hashmap
for each element in array:
    if complement in hashmap:
        return indices
    else:
        store element in hashmap
```

---

3. Two-Pointers

Pattern: Use two indices moving inward/outward to meet certain conditions, O(N) or O(N log N).

Strategies:

- For sorted arrays, move left/right pointers inward to meet sum or difference.
- For partitioning or rearranging in-place.
- When asked for pair with sum closest to target.
- In string scanning problems (palindromes, container with most water).

Watch for: proper increment/decrement rules, skipping duplicates, initializing pointers.

Archetypes:

- 2-sum II input sorted
- Container With Most Water
- Sort colors (0/1/2) partitioning
- Valid palindrome (skip non-alnums)

Recognition cues: sorted data, sum/difference target among two elements, palindrome check.

Technique Overview:
Used with **sorted arrays or strings** to efficiently find **pairs, subsequences, or to partition data**.

Pseudocode Template:

```
left = 0
right = n - 1
while left < right:
    if condition met:
        update result
        move pointers appropriately
    else if need to increase sum:
        move left pointer
    else:
        move right pointer
```

---

4. Stacks

Pattern: LIFO processing, suitable for matching, backtracking, monotonic sequences.

Strategies:

- Use stack for parentheses balancing.
- Monotonic stack for next greater/smaller elements.
- Use stack instead of recursion (e.g., DFS manual).
- Evaluate expression with operator stack + operand stack.

Watch for: stack underflow, empty-stack edge cases, clearing stack at end.

Archetypes:

- Valid Parentheses
- Evaluate Reverse Polish Notation
- Largest Rectangle in Histogram (monotonic)
- Next Greater Element

Recognition cues: phrase like “next greater”, “balanced”, “evaluate”, “parentheses”, “monotonic”.

Technique Overview:
Stacks help in **tracking previous elements**, **balancing parentheses**, or **monotonic patterns**.

Pseudocode Template:

```
initialize stack
for each element in input:
    while stack not empty and stack.top invalid compared to element:
        pop stack
    push element to stack
process remaining stack
```

---

5. Linked Lists

Pattern: Singly/doubly linked nodes, pointers, no index access.

Strategies:

- Use dummy head to handle head changes cleanly.
- Fast & slow pointers for cycle detection, middle.
- Reversals (whole or part) via pointer rewiring.
- Merge lists with two-pointer stepping.
- Recursion for list processing (e.g., reverse recursively).

Watch for: null pointer, next vs. prev confusion, infinite loops, memory (don’t create cycles).

Archetypes:

- Remove N-th node from end
- Reverse Linked List (iteratively/recursively)
- Detect Cycle / Find cycle start (Floyd’s tortoise and hare)
- Merge two sorted lists
- Copy List with Random Pointer

Recognition cues: mentions “list”, “node”, “next”, “dummy”, “cycle”, “reverse”.

Technique Overview:
Linked lists rely on **pointer manipulation**, often with **dummy nodes** to simplify edge cases.

Pseudocode Template:

```
dummy = new node
dummy.next = head
prev = dummy
current = head
while current:
    process node
    current = current.next
return dummy.next
```

---

6. Binary Search

Pattern: Search on sorted data or monotonic function domain.

Strategies:

- Find exact match or boundary (first ≥ target, last ≤ target).
- Use left ≤ right or left < right loops consistently.
- Handle infinite loop by mid = left + (right-left)/2.
- Apply to index search or value search (conceptual monotonic).

Watch for: off-by-one, mid rounding, loop condition errors.

Archetypes:

- Standard binary search (find target)
- Find first/last occurrence
- Search insert position
- Peak index in mountain array
- kth smallest in sorted matrix (with binary search on value)

Recognition cues: sorted input, asking for “position” or “range”, target threshold.

Technique Overview:
Use when data is **sorted** or when **decision functions are monotonic**.

Pseudocode Template:

```
left = 0
right = n - 1
while left <= right:
    mid = left + (right - left) // 2
    if condition(mid):
        return mid
    else if need smaller:
        right = mid - 1
    else:
        left = mid + 1
return -1
```

---

7. Sliding Window

Pattern: Maintain a window [left, right] with dynamic size to satisfy a condition. Often O(N).

Strategies:

- Expand right until condition satisfied, then shrink left.
- Track state: count of elements, sum, max/min, distinct count, etc.
- For fixed window size: just iterate with moving boundaries.

Watch for: updating counts correctly when moving left and right, empty window edge.

Archetypes:

- Longest substring with at most K distinct chars
- Minimum window substring (s containing t)
- Subarray sum equals k (if nonnegative; or prefix + hash for general)
  Max sum subarray of size k

Recognition cues: phrases like “subarray substring”, “longest”, “shortest”, “at most”, “contains”, dynamic window length.

Technique Overview:
Maintain a **window [left, right]** and adjust it dynamically based on constraints.

Pseudocode Template:

```
left = 0
for right in range(n):
    add current element to window
    while window invalid:
        shrink from left
    update result with current window
```

---

8. Trees

Pattern: Hierarchical, branching, recursive by nature (often binary tree).

Strategies:

- Use DFS (pre/in/post) or BFS (level-order via queue).
- Recursion returns useful info; bottom-up aggregation.
- Track path for root-to-leaf tasks.
- Use stack or queue to simulate traversal iteratively.

Watch for: null child checks, deep recursion depth, off-by-one in levels.

Archetypes:

- Invert binary tree
- Maximum depth / minimum depth
- Validate binary search tree (in-order vs. bounds)
- Level order traversal / zigzag level order
- Path sum / all paths / sum of left leaves

Recognition cues: a tree node given; traversal, sum, depth, path tasks.

Technique Overview:
Trees often require **DFS (recursive)**, **BFS (queue)**, or **post-order traversals** for bottom-up calculations.

DFS Pseudocode:

```
function dfs(node):
    if node is null:
        return
    dfs(node.left)
    dfs(node.right)
    process node
```

BFS Pseudocode:

```
queue = [root]
while queue not empty:
    size = len(queue)
    for i in range(size):
        node = queue.pop(0)
        process node
        if node.left: queue.push(node.left)
        if node.right: queue.push(node.right)
```

---

9. Heaps / Priority Queue

Pattern: Always extract or peek min or max; maintain top-k; merging sorted streams.

Strategies:

- Use heap to keep k largest/smallest elements.
- Use heap to merge k sorted lists.
- Use heap to simulate “sliding k-window max”.
- Use min-heap for “meeting rooms” or “task scheduler”.

Watch for: heap size management, custom comparator, large k vs n.

Archetypes:

- Kth largest element in an array
- Merge k sorted lists
- Sliding window maximum
- Task scheduler with cooldown

Recognition cues: “top k”, “merge k”, “meeting rooms”, “schedule”, “sliding window max”.

Technique Overview:
Use heaps for **top-k problems**, **dynamic streams**, or **efficiently merging**.

Pseudocode Template:

```
initialize min_heap
for each element:
    push element into heap
    if heap size > k:
        pop smallest
return heap
```

---

10. Recursive / Backtracking

Pattern: Explore all valid combinations/paths; decision tree.

Strategies:

- Define backtrack function with current path and remaining choices.
- Base case: path forms valid answer → record.
- Loop choices, take action, recurse, undo (backtrack).
- Prune early when current state violates constraints.

Watch for: deep recursion stack, duplicates (use visited or sorting/pruning), mutable state.

Archetypes:

- Subsets / permutations / combinations
- Word search / permute unique
- N-Queens
- Sudoku solver
- Restore IP addresses

Recognition cues: words like “all”, “combinations”, “permutations”, “subset”, “restore”, “generate”, “solve”.

Technique Overview:
Explore **all combinations or paths**; backtrack after each recursive call.

Pseudocode Template:

```
function backtrack(path, choices):
    if path is a solution:
        add to results
        return
    for choice in choices:
        make choice
        backtrack(path + choice, remaining choices)
        undo choice
```

---

11. Graphs

Pattern: Nodes with edges; may be directed or undirected; may be weighted.

Strategies:

- BFS for shortest unweighted distance, level traversal.
- DFS for connectivity, cycle detection, topological sort.
- Dijkstra for weighted shortest path (non-negative weights).

- Union-find for connectivity in dynamic edge addition.
- Adjacency list representation.

Watch for: visited marking, cycles, directed vs undirected nuance, large recursion depth.

Archetypes:

- Number of Islands (grid as graph)
- Course Schedule (cycle detection / topo sort)
- Word Ladder (BFS)
- Minimum Spanning Tree / network connectivity (Union-Find / Kruskal)
- Shortest path (Dijkstra)

Recognition cues: grid, nodes/edges, connectivity, shortest, cycle, schedule.

Technique Overview:
Use **BFS for shortest path**, **DFS for traversal**, **Union-Find for connectivity**, or **Dijkstra for weighted shortest path**.

BFS Pseudocode:

```
queue = [start]
visited = set([start])
while queue:
    node = queue.pop(0)
    process node
    for neighbor in graph[node]:
        if neighbor not in visited:
            visited.add(neighbor)
            queue.append(neighbor)
```

DFS Pseudocode:

```
function dfs(node):
    if node in visited:
        return
    visited.add(node)
    for neighbor in graph[node]:
        dfs(neighbor)
```

---

12. Dynamic Programming (DP)

Pattern: Optimal solution built from subproblem solutions; overlapping subproblems, optimal substructure.

Strategies:

- Identify state and state-transition.
- Choose bottom-up (array/table) or top-down (memoization).
- Typical dimensions: index, remaining capacity, partition count, previous decision, etc.
- For subsequence/subarray: track by index and inclusion/exclusion.
- For knapsack/partition: 1D or 2D DP.

Watch for: memory/time large DP, choosing right dimension/order, boundary values, initialization base case.

Archetypes:

- Fibonacci / climb stairs
- House robber / house robber II
- Coin change / unbounded knapsack
- Longest increasing subsequence
- Word break
- Decode ways
- Jump game

Recognition cues: “maximum”, “minimum” over substructures, “ways to”, “longest”, “shortest”, “can you”, “count number of”, overlapping decisions.

Technique Overview:
Define **state and transitions**; use **bottom-up or top-down** memoization.

Pseudocode Template:

```
dp = initialize array
for i in range(n):
    for each state dependent on i:
        dp[i] = min/max(dp[i], transition)
return dp[target]
```

---

13. Bit Manipulation

Pattern: Use bits for flags, parity, counting ones, shifting, masks.

Strategies:

- XOR to toggle/invert or find odd-one-out.
- Bitwise AND to test flags.
- Use x & (x − 1) to drop lowest set bit.
- Use shifts for division/multiplication or scanning bits.
- Precompute bit counts or use built-ins.

Watch for: signed vs unsigned shifts, overflow, word size limits.

Archetypes:

- Single Number (find unique with XOR)
- Number of 1 Bits (popcount)
- Power of Two / Power of Three using bit pattern.
- Subsets using bitmask enumeration.

Recognition cues: talk of bits, binary strings, powers of two, flags, masks.

Technique Overview:
Use **bitwise operators** for masks, toggling, or set checks.

Pseudocode Template:

```
for i from 0 to 31:
    if (num >> i) & 1:
        process bit
```

---

14. Math

Pattern: Use arithmetic properties: divisibility, primes, factorization, geometry.

Strategies:

- Use basic loops up to sqrt(n).
- Sieve of Eratosthenes for prime lists.
- Modular arithmetic to avoid overflow.
- Use geometric formulas (area, distance).
- Recognize patterns (triangular numbers, factorial, combinations).

Watch for: integer overflow, floating-point precision, off-by-one in loops.

Archetypes:

- Count Primes
- Perfect Number / Abundant or Deficient

- Sqrt(x) integer class
- GCD / LCM / Euclidean algorithm
- Happy Number

Recognition cues: ask for primes, perfect numbers, mathematical property, integer math not structural.

Technique Overview:
Use **formulas, number theory, and modular arithmetic** to reduce complexity.

Pseudocode Template:

```
result = 0
for i in range(1, sqrt(n) + 1):
    if n % i == 0:
        process factor
```

---

15. Greedy

Pattern: Make local best choice hoping it leads to global optimum, O(N) or O(N log N).

Strategies:

- Sort and then pick based on criteria.
- Always pick the earliest finishing interval (interval scheduling).
- Always pick smallest/largest valid choice and move on.
- Validate greedy via “exchange argument” or proof.

Watch for: greed may fail if not validated; watch edge cases.

Archetypes:

- Jump Game (greedy furthest reach)
- Interval Scheduling (meeting room / erase overlap intervals)
- Gas Station / Minimum platforms
- Candy / lemonade change

Recognition cues: ask for minimum number of, maximum number, feasibility over segments, “can you”, “minimum steps”.

Technique Overview:
Choose the **best local option** at each step; sort data if needed to guide selection.

Pseudocode Template:

```
sort array by criteria
for each element in array:
    if condition:
        take element
    update result
```

---

16. Memoization (Top-Down DP)

Concept

- Recursive approach with **caching**.
- Start with the main problem and **break it down into subproblems**.
- Store (memoize) results of solved subproblems to avoid recomputation.
- Typically implemented with recursion + a dictionary/array for storage.

Key Steps

1. Define the recursive function that represents the problem state.
2. Check if the result is already stored in the cache (memo).
3. If not computed, solve recursively and store the result in the memo.
4. Return the stored result.

Pseudocode

```
function dp(state):
    if state in memo:
        return memo[state]
    if base_case(state):
        return base_result
    result = combine(dp(next_state1), dp(next_state2), ...)
    memo[state] = result
    return result
return dp(initial_state)
```

Example

Problem: Climbing Stairs (#70)
You can climb 1 or 2 steps at a time. Find the number of distinct ways to reach the top.

```
function climb(n):
    if n in memo:
        return memo[n]
    if n <= 2:
        return n
    memo[n] = climb(n-1) + climb(n-2)
    return memo[n]
return climb(n)
```

When to Use

- When the recursive relationship is natural and easy to express.
- When only a small portion of the state space is explored.

Pros

- Easy to implement.
- Great for problems with a naturally recursive structure.

Cons

- Recursive call stack can cause stack overflow in deep recursion.
- Slightly slower due to recursive overhead compared to tabulation.

---

17. Tabulation (Bottom-Up DP)

Concept

- Iterative approach that builds the solution **from smaller subproblems up to the final answer**.
- Typically uses an array or table to store results of subproblems.

Key Steps

1. Identify the base cases.
2. Initialize a DP table with base values.
3. Fill the table iteratively according to the state transitions.
4. Return the final value from the table.

Pseudocode

```
dp = array of size n+1
dp[0] = base_case_0
dp[1] = base_case_1
```

```
for i from 2 to n:
    dp[i] = combine(dp[i-1], dp[i-2], ...)

return dp[n]
```

Example

Problem: Climbing Stairs (#70)

```
dp[0] = 1
dp[1] = 1
for i from 2 to n:
    dp[i] = dp[i-1] + dp[i-2]
return dp[n]
```

When to Use

- When you need better control over time and space usage.
- When recursion depth could cause stack overflows.

Pros

- Faster in practice (no recursion overhead).
- Easy to optimize space (e.g., keep only the last few states).

Cons

- Requires knowing the iteration order to fill the table correctly.
- Slightly more complex to derive the iteration order compared to memoization.

Quick Reference Table

Technique	Recognize When...	Key Idea
Arrays	direct indexing, contiguous storage	sort, prefix sums, two-pointer
HashMap / Set	fast lookup, counting, complements	mapping count or presence
Two-Pointers	sorted or pairing, search by sum/diff	move pointers based on sum condition

Stack	LIFO pattern, matching, monotonic sequence	push/pop operations
Linked Lists	`next`, `prev`, cycle, merge, reverse	pointer manipulation, dummy nodes
Binary Search	sorted data or monotonic decision	bisect to find target or boundary
Sliding Window	contiguous subsequence with constraints	expand/shrink window dynamically
Tree	hierarchical node structure	DFS/BFS, recursion, tree depth/paths
Heap / PQ	top-k, merging, ordering demands	maintain heap and extract min/max
Recursion/Backtr.	generate all combinations/paths	explore choices, backtrack on return
Graph	nodes and edges, connectivity, cycles	BFS/DFS, Dijkstra, Union-Find, topo sort
DP	overlapping subproblems, optimal substructure	define state, transitions, memo/table
Bit Manipulation	problems about bits, parity, toggles	XOR, shifts, masks
Math	numeric properties, divisibility, primes	number theory, iteration, formulas
Greedy	local optimum choice building global solution	pick best step greedily, prove correctness

---

How to Use When Facing a New Problem

1. **Classify by key clues**:
    - Data structure (array, tree, graph, list).
    - Ask what problem type (count, search, optimize, decision).
2. **Pick candidate technique(s)**: sometimes two overlap (e.g., sliding window + hashmap).
3. **Sketch the approach**: define pointers/indices/recurrences, outline transitions.
4. **Check edge cases**: empty input, single element, large ranges, overlaps.
5. **Estimate complexity**: can you do better than brute force? What are time/space costs?
6. **Refine**: add pruning, use auxiliary structures, convert recursion to iteration if needed.
