LeetCode Cheat Sheet

Big O Complexity (Fastest to Slowest)

Complexity	Name	Example Operations
O(1)	Constant	Array access, hash table lookup, push/pop
O(log n)	Logarithmic	Binary search, balanced BST operations
O(n)	Linear	Single loop, linear search, two pointers
O(n log n)	Linearithmic	Merge sort, heap sort, efficient sorting
O(n^2)	Quadratic	Nested loops, bubble sort, insertion sort
O(n^3)	Cubic	Triple nested loops, naive matrix multiplication
O(2^n)	Exponential	Recursive subsets, naive fibonacci
O(n!)	Factorial	Generating all permutations

Constraint Ranges -> Expected Complexity

Constraint	Target Complexity	Typical Approach
n <= 10	O(n!) or O(2^n)	Brute force, permutations, backtracking
n <= 20	O(2^n)	Backtracking, bitmask DP
n <= 100	O(n^3)	Triple loops, Floyd-Warshall
n <= 1,000	O(n^2)	Nested loops, 2D DP
n <= 10,000	O(n^2) (borderline)	Optimized n^2, consider n log n
n <= 100,000	O(n log n)	Sorting, heap, binary search
n <= 1,000,000	O(n)	Hash map, two pointers, sliding window
n <= 10,000,000+	O(n) or O(log n)	Math, binary search, O(1) tricks

Pattern Recognition -> Data Structure / Algorithm

Arrays & Strings

Problem Pattern	Data Structure / Algorithm
Find pair with target sum	Hash map, Two pointers (sorted)
Find subarray with target sum	Prefix sum + Hash map, Sliding window
Maximum/minimum subarray	Kadane's algorithm, DP
Sorted array operations	Binary search, Two pointers
Merge sorted arrays	Two pointers
k-th largest/smallest	Heap, Quickselect
Top k frequent elements	Heap, Bucket sort
Sliding window max/min	Monotonic deque
Subarray/substring with constraint	Sliding window
Longest increasing subsequence	DP, Binary search + patience sort
Range sum queries	Prefix sum, Segment tree

Linked Lists

Problem Pattern	Technique
Find middle	Slow/fast pointers
Detect cycle	Floyd's algorithm (slow/fast)
Find cycle start	Floyd's phase 2
Reverse list	Iterative (prev, curr, next)
Merge two lists	Two pointers, dummy head
k-th from end	Two pointers (k gap)
Reorder list	Find middle + reverse + merge

Trees

Problem Pattern	Technique
Traversal (in/pre/post)	Recursion, Stack (iterative)
Level order	BFS with queue
Path sum problems	DFS with running sum
Lowest common ancestor	Recursive DFS

Validate BST	Inorder traversal, or min/max bounds
Serialize/deserialize	BFS or preorder + null markers
Diameter of tree	DFS, track max(left + right)
Count nodes, height	Recursive DFS

Graphs

Problem Pattern	Algorithm
Traversal, connected components	BFS, DFS
Shortest path (unweighted)	BFS
Shortest path (weighted, positive)	Dijkstra's
Shortest path (negative weights)	Bellman-Ford
All pairs shortest path	Floyd-Warshall
Detect cycle (directed)	DFS with colors (white/gray/black)
Detect cycle (undirected)	Union-Find, DFS with parent tracking
Topological sort	Kahn's (BFS), or DFS post-order
Minimum spanning tree	Prim's, Kruskal's
Find connected components	Union-Find, DFS/BFS
Bipartite check	BFS/DFS with 2-coloring
Number of islands	DFS/BFS flood fill

Dynamic Programming

Problem Pattern	DP Type
Fibonacci-style	1D DP, O(1) space optimization
Coin change, climbing stairs	1D DP
Knapsack (bounded/unbounded)	1D or 2D DP
Longest common subsequence	2D DP
Edit distance	2D DP
Matrix path problems	2D DP
Palindrome problems	2D DP (i, j range)

Partition problems	2D DP, subset sum
String matching (regex, wildcard)	2D DP
Stock buy/sell with constraints	State machine DP
Interval scheduling	DP + sorting

Intervals

Problem Pattern	Technique
Merge overlapping	Sort by start, merge
Insert interval	Binary search or linear merge
Meeting rooms (can attend all?)	Sort, check overlap
Meeting rooms II (min rooms)	Sort + heap, or sweep line
Non-overlapping intervals	Greedy, sort by end

Stack & Queue

Problem Pattern	Data Structure
Valid parentheses	Stack
Next greater/smaller element	Monotonic stack
Largest rectangle in histogram	Monotonic stack
Min stack	Stack with min tracking
Sliding window maximum	Monotonic deque
Evaluate expressions	Stack (operands + operators)
Decode strings	Stack

Heap (Priority Queue)

Problem Pattern	Technique
k-th largest	Min heap of size k
k-th smallest	Max heap of size k
Merge k sorted lists	Min heap
Top k frequent	Min heap + frequency map

Median from stream	Two heaps (max + min)
Task scheduler	Max heap + cooldown
Meeting rooms II	Min heap for end times

Binary Search

Problem Pattern	Technique
Search in sorted array	Standard binary search
Search in rotated array	Modified binary search
Find first/last occurrence	Binary search with bias
Search insert position	Lower bound
Peak element	Binary search on answer
Minimum in rotated array	Binary search
Capacity/allocation problems	Binary search on answer
Square root, nth root	Binary search on answer

Backtracking

Problem Pattern	Template
Generate all subsets	Include/exclude each element
Generate all permutations	Swap or used[] array
Combination sum	Backtrack with running sum
N-Queens	Row by row placement
Sudoku solver	Cell by cell with validation
Word search in grid	DFS with visited marking
Palindrome partitioning	Backtrack with palindrome check

Greedy

Problem Pattern	Approach
Interval scheduling	Sort by end time
Jump game	Track max reachable

Gas station	Track running surplus
Task scheduler	Fill gaps based on max frequency
Reorganize string	Greedy heap placement
Candy distribution	Two pass (left-to-right, right-to-left)

Key Data Structure Operations

Hash Map / Hash Set

```
Insert: O(1) average
Lookup: O(1) average
Delete: O(1) average
```

Use when: Need fast lookup, counting frequencies, finding pairs/complements

Heap (Priority Queue)

```
Insert: O(log n)
Extract min/max: O(log n)
Peek: O(1)
Heapify array: O(n)
```

Use when: Need repeated min/max extraction, k-th element problems

Stack

```
Push: O(1)
Pop: O(1)
Peek: O(1)
```

Use when: LIFO needed, matching brackets, monotonic problems, DFS

Queue / Deque

```
Enqueue/Dequeue: O(1)
```

Use when: FIFO needed, BFS, sliding window

Binary Search Tree (Balanced)

```
Insert: O(log n)
Delete: O(log n)
Search: O(log n)
```

Use when: Need sorted order with fast insert/delete

Trie

```
Insert: O(m) where m = word length
Search: O(m)
Prefix search: O(m)
```

Use when: Prefix matching, autocomplete, word dictionaries

Union-Find (Disjoint Set)

```
Find: O(a(n)) ~ O(1) with path compression
Union: O(a(n)) ~ O(1) with union by rank
```

Use when: Connected components, cycle detection in undirected graphs

---

Common Formulas & Tricks

Two Pointers

- **Opposite ends:** Start at 0 and n-1, move inward
- **Same direction:** Slow/fast, or sliding window
- **Sorted array:** Often enables O(n) solutions

Sliding Window Template

```
left = 0
for right in range(len(arr)):
   # expand window: add arr[right] to state

   while window_invalid():
       # shrink window: remove arr[left] from state
       left += 1

   # update answer with current window
```

Binary Search Template

```
left, right = 0, len(arr) - 1
while left <= right:
   mid = left + (right - left) // 2
   if arr[mid] == target:
       return mid
   elif arr[mid] < target:
       left = mid + 1
   else:
       right = mid - 1
return -1 # or left for insert position
```

DFS Template (Recursive)

```
def dfs(node):
   if not node:
       return
   # process node
   dfs(node.left)
   dfs(node.right)
```

BFS Template

```
from collections import deque
queue = deque([start])
visited = {start}
while queue:
   node = queue.popleft()
   for neighbor in get_neighbors(node):
       if neighbor not in visited:
           visited.add(neighbor)
           queue.append(neighbor)
```

Backtracking Template

```
def backtrack(path, choices):
   if is_solution(path):
       result.append(path[:])
       return

   for choice in choices:
       if is_valid(choice):
           path.append(choice)
           backtrack(path, remaining_choices)
           path.pop() # undo choice
```

Floyd's Cycle Detection

```
# Phase 1: Find meeting point
slow = fast = head
while fast and fast.next:
   slow = slow.next
   fast = fast.next.next
   if slow == fast:
       break

# Phase 2: Find cycle start
slow = head
while slow != fast:
   slow = slow.next
   fast = fast.next
return slow # cycle entrance
```

Prefix Sum

```
prefix = [0]
for num in arr:
   prefix.append(prefix[-1] + num)

# Sum of arr[i:j] = prefix[j] - prefix[i]
```

Monotonic Stack (Next Greater Element)

```
result = [-1] * len(arr)
stack = [] # stores indices
for i, num in enumerate(arr):
   while stack and arr[stack[-1]] < num:
       result[stack.pop()] = num
   stack.append(i)
```

Edge Cases to Always Consider

- Empty input (`[]`, `""`, `None`)
- Single element
- Two elements
- All same elements
- Already sorted / reverse sorted
- Negative numbers
- Integer overflow (use `left + (right - left) // 2`)
- Duplicates
- Odd vs even length

Python Tips for LeetCode

```
# Infinity
float('inf'), float('-inf')

# Integer limits
import sys
sys.maxsize # max int

# Default dict
from collections import defaultdict
d = defaultdict(int) # default 0
d = defaultdict(list) # default []

# Counter
from collections import Counter
freq = Counter("aabbc") # {'a': 2, 'b': 2, 'c': 1}
freq.most_common(2) # [('a', 2), ('b', 2)]

# Heap (min heap by default)
import heapq
heapq.heappush(heap, val)
heapq.heappop(heap)
heapq.heapify(arr) # in-place O(n)
# Max heap: push -val, negate on pop

# Deque
from collections import deque
dq = deque()
dq.append(x) # right
dq.appendleft(x) # left
dq.pop() # right
dq.popleft() # left

# Binary search
from bisect import bisect_left, bisect_right
i = bisect_left(arr, x) # leftmost position to insert x
i = bisect_right(arr, x) # rightmost position to insert x

# Sort with key
arr.sort(key=lambda x: x[1]) # sort by second element
arr.sort(key=lambda x: (-x[0], x[1])) # descending first, ascending second

# List comprehension with condition
[x for x in arr if x > 0]

# Initialize 2D array
```

```
dp = [[0] * cols for _ in range(rows)]
# NOT: [[0] * cols] * rows (creates references!)

# Swap
a, b = b, a

# Ceiling division
-(-a // b) # or math.ceil(a / b)
```