Amazon SDE-1 Interview Cheat Sheet

Quick Reference for Interview Success

Big O Notation Quick Reference

Complexity	Name	Example
O(1)	Constant	Hash table lookup, array access
O(log n)	Logarithmic	Binary search, balanced BST
O(n)	Linear	Array traversal, linear search
O(n log n)	Log-linear	Merge sort, heap sort
O(n²)	Quadratic	Bubble sort, nested loops
O(2^n)	Exponential	Recursive fibonacci (naive)

Data Structures - Time & Space Complexity

Array

Operation	Average	Worst	Space
Access	O(1)	O(1)	O(n)
Search	O(n)	O(n)	O(1)
Insert	O(n)	O(n)	O(1)
Delete	O(n)	O(n)	O(1)

Key Points:

- Static vs Dynamic arrays
- Use for: Lookups, math operations, cache-friendly
- Avoid for: Middle insertions/deletions

Hash Table

Operation	Average	Worst	Space
Search	O(1)	O(n)	O(n)
Insert	O(1)	O(n)	O(n)
Delete	O(1)	O(n)	O(n)

Key Points:

- Handle collisions: Chaining, Open Addressing
- Load factor should be < 0.75
- Use for: Fast lookups, caching, frequency counting

Linked List

Operation	Time	Space
Access	O(n)	O(1)
Search	O(n)	O(1)
Insert	O(1)	O(1)
Delete	O(1)	O(1)

Key Points:

- Singly vs Doubly linked
- Good for: Dynamic size, frequent insertions
- Bad for: Random access, extra memory overhead

Stack & Queue

Operation	Stack	Queue	Space
Insert	O(1)	O(1)	O(1)
Delete	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)

Stack Uses: DFS, expression parsing, undo operations
Queue Uses: BFS, process scheduling, breadth-first problems

Binary Search Tree

Operation	Average	Worst	Space
Search	O(log n)	O(n)	O(n)
Insert	O(log n)	O(n)	O(n)
Delete	O(log n)	O(n)	O(n)

Key Points:

- Balanced: AVL, Red-Black trees
- In-order traversal gives sorted order
- Good for: Range queries, sorted data

Heap

Operation	Time	Space
Insert	O(log n)	O(n)
Delete Max/Min	O(log n)	O(n)
Get Max/Min	O(1)	O(n)

Use for: Priority queues, top K problems, heap sort

Essential Algorithm Patterns

1. Two Pointers

```
def two_pointer_template(arr):
    left, right = 0, len(arr) - 1
    while left < right:
        # Process current pair
        if condition:
            left += 1
        else:
            right -= 1
```

Use for: Two Sum, Palindromes, Sorted arrays

2. Sliding Window

```
def sliding_window_template(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i-k]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

Use for: Subarray problems, string patterns

3. Fast & Slow Pointers

```
def detect_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
```

```
            return True
    return False
```

Use for: Cycle detection, finding middle element

4. Binary Search Template

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Use for: Sorted arrays, search space problems

5. DFS Template

```
def dfs(node, visited):
    if node in visited:
        return
    visited.add(node)
    # Process node
    for neighbor in node.neighbors:
        dfs(neighbor, visited)
```

6. BFS Template

```
from collections import deque

def bfs(start):
    queue = deque([start])
    visited = set([start])

    while queue:
        node = queue.popleft()
        # Process node
        for neighbor in node.neighbors:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Dynamic Programming Patterns

1. Fibonacci Pattern

```
def fibonacci(n):
    if n <= 1: return n
    prev2, prev1 = 0, 1
    for i in range(2, n + 1):
        current = prev1 + prev2
        prev2, prev1 = prev1, current
    return prev1
```

2. 0/1 Knapsack Pattern

```
def knapsack(weights, values, capacity):
    dp = [[0] * (capacity + 1) for _ in range(len(weights) + 1)]

    for i in range(1, len(weights) + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(
                    dp[i-1][w],  # Don't take
                    values[i-1] + dp[i-1][w - weights[i-1]]  # Take
                )
            else:
                dp[i][w] = dp[i-1][w]
    return dp[len(weights)][capacity]
```

Common Tricks & Techniques

Hash Map for O(1) Lookups

```
# Two Sum using hash map
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
```

Dummy Node for Linked Lists

```
def merge_lists(l1, l2):
    dummy = ListNode(0)
    current = dummy

    while l1 and l2:
```

```
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    current.next = l1 or l2
    return dummy.next
```

Use Built-in Data Structures

```
import heapq
from collections import deque, defaultdict, Counter

# Priority Queue (Min Heap)
heap = []
heapq.heappush(heap, item)
min_item = heapq.heappop(heap)

# Counter for frequency
freq = Counter(array)
```

Interview Best Practices

Before Coding:

1. **Understand the problem** - Ask clarifying questions
2. **Think about edge cases** - Empty inputs, single elements
3. **Discuss approach** - Brute force first, then optimize
4. **Choose data structure** - Based on operations needed
5. **Estimate complexity** - Time and space

While Coding:

1. **Start with brute force** if optimal solution isn't obvious
2. **Use descriptive variable names**
3. **Add comments for complex logic**
4. **Handle edge cases**
5. **Think out loud**

After Coding:

1. **Test with examples** - Walk through your code
2. **Check edge cases**
3. **Analyze time/space complexity**
4. **Discuss potential optimizations**

Amazon Leadership Principles Focus:

- **Customer Obsession** - Think about user impact
- **Ownership** - Take responsibility for the solution
- **Invent and Simplify** - Find elegant solutions
- **Dive Deep** - Understand the problem thoroughly

Most Asked Amazon Questions (Quick List)

Easy (Master These First):

- Two Sum (#1)
- Valid Parentheses (#20)
- Best Time to Buy and Sell Stock (#121)
- Maximum Depth of Binary Tree (#104)
- Climbing Stairs (#70)

Medium (Core Amazon Questions):

- Trapping Rain Water (#42)
- Number of Islands (#200)
- LRU Cache (#146)
- Product of Array Except Self (#238)
- Coin Change (#322)
- Course Schedule (#207)
- Group Anagrams (#49)

Hard (Advanced):

- Merge k Sorted Lists (#23)
- LFU Cache (#460)
- Edit Distance (#72)

Last Minute Tips

Time Management:

- 40-45 minutes total for coding questions
- 5 minutes: Understanding + clarification
- 5 minutes: Approach discussion
- 25 minutes: Coding
- 5 minutes: Testing + optimization

When Stuck:

1. Start with brute force
2. Look for patterns in examples
3. Consider different data structures
4. Think about edge cases
5. Ask for hints (it's okay!)

Red Flags to Avoid:

- Jumping into code without discussion
- Not handling edge cases
- Silent coding
- Not testing your solution
- Giving up too easily

Green Flags to Show:

- Clear communication
- Systematic approach
- Code organization
- Edge case consideration
- Optimization thinking

Remember: Amazon values problem-solving process over perfect solutions. Show your thinking, communicate clearly, and demonstrate the leadership principles throughout your approach!

Good Luck!