SNAP (Snapchat) Backend Engineer Interview Preparation

Comprehensive Problem Set with Solutions

2025

Contents

1 Introduction

1.1 About SNAP Inc.

SNAP Inc. (founded 2011, Nasdaq: SNAP) is a camera company that created Snapchat, a multimedia messaging app with 400M+ daily active users worldwide. SNAP focuses on:

- **Ephemeral Content**: Photos and videos that disappear after viewing
- **AR Technology**: Advanced face filters and lenses using computer vision
- **Stories**: 24-hour temporary content timelines
- **Spotlight**: Short-form video discovery platform
- **Real-time Communication**: Low-latency messaging and video streaming

1.2 Interview Process

Based on actual candidate experiences:

1. **Recruiter Screen** (30 min): Background, interest in camera/AR/social products
2. **Technical Phone Screen** (45-60 min): 1-2 LeetCode medium problems, expects *runnable code* (no pseudocode)
3. **Onsite/Virtual** (4-6 hours):
   - 2-4 Coding Rounds: LeetCode medium/hard, emphasis on speed and correctness
   - 1-2 System Design: Snapchat features or scalable infrastructure
   - Behavioral: Integrated throughout, values "Kind, Smart, Creative"

1.3 Key Technical Focus Areas

- **Graphs**: BFS/DFS, shortest path (Dijkstra, A*), grid traversal
- **Linked Lists**: Deep copy with random pointers, reversal
- **Trees**: Serialization/deserialization, traversals
- **Arrays/Stacks**: Rainwater trapping, monotonic stack
- **Hash Maps**: LRU cache, frequency counting
- **Heaps**: Priority queues, top K, scheduling
- **System Design**: CDN, real-time messaging, ephemeral content, microservices

1.4 Common Interview Questions (Actual)

From Glassdoor, LeetCode, , and candidate reports:

- Shortest Path in Maze (with wall breaking)
- Copy List with Random Pointers
- Serialize and Deserialize Binary Tree
- Trapping Rainwater
- LRU Cache
- Number of Islands (variations)
- Design Stories feature / Real-time messaging / AR filter system

2 Problem 1: Shortest Path in Maze

2.1 Problem Description

SNAP commonly asks graph/maze problems related to video games and AR navigation features. This problem appears frequently in their phone screens and onsite interviews.

Given: A 2D grid where 0 represents walkable path and 1 represents wall/obstacle.

Tasks:

1. Find the shortest path from start to end position
2. Support wall-breaking (cost = 1 per wall, walk cost = 1)
3. Return minimum distance or -1 if no path exists

Example:

```
grid = [
    [0, 0, 1, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1],
    [1, 0, 0, 0]
]
start = (0, 0), end = (3, 3)

shortest_path(grid, start, end)  # Returns: 6
shortest_path_with_breaking(grid, start, end)  # Returns: 5
```

Constraints:

- $1 \leq$ rows, cols ≤ 100
- grid[i][j] $\in \{0, 1\}$
- start and end are valid positions

2.2 Solution

Approach 1: BFS for Shortest Path (No Wall Breaking)

```python
from collections import deque
from typing import List, Tuple

def shortest_path(grid: List[List[int]],
                  start: Tuple[int, int],
                  end: Tuple[int, int]) -> int:
    """BFS to find shortest path without breaking walls."""
    if not grid or not grid[0]:
        return -1

    rows, cols = len(grid), len(grid[0])
    if grid[start[0]][start[1]] == 1 or grid[end[0]][end[1]] == 1:
        return -1

    queue = deque([(start[0], start[1], 0)])
    visited = {start}
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    while queue:
        r, c, dist = queue.popleft()

        if (r, c) == end:
            return dist

        for dr, dc in directions:
            nr, nc = r + dr, c + dc
```

```python
            if (0 <= nr < rows and 0 <= nc < cols and
                grid[nr][nc] == 0 and (nr, nc) not in visited):
                visited.add((nr, nc))
                queue.append((nr, nc, dist + 1))

    return -1
```

Approach 2: Dijkstra for Minimum Cost (With Wall Breaking)

```python
import heapq

def shortest_path_with_breaking(grid: List[List[int]],
                                start: Tuple[int, int],
                                end: Tuple[int, int]) -> int:
    """Dijkstra's algorithm allowing wall breaking."""
    if not grid or not grid[0]:
        return -1

    rows, cols = len(grid), len(grid[0])
    heap = [(0, start[0], start[1])]  # (cost, row, col)
    visited = {}
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    while heap:
        cost, r, c = heapq.heappop(heap)

        if (r, c) == end:
            return cost

        if (r, c) in visited and visited[(r, c)] <= cost:
            continue
        visited[(r, c)] = cost

        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols:
                # Cost: 1 for walk, 1 for breaking wall
                new_cost = cost + 1
                if (nr, nc) not in visited or visited[(nr, nc)] > new_cost:
                    heapq.heappush(heap, (new_cost, nr, nc))

    return -1
```

2.3 Complexity Analysis

BFS Approach:

- **Time**: $O(R \times C)$ - visit each cell at most once
- **Space**: $O(R \times C)$ - queue and visited set

Dijkstra Approach:

- **Time**: $O(R \times C \times \log(R \times C))$ - heap operations
- **Space**: $O(R \times C)$ - heap and visited map

3 Problem 2: Ephemeral Message Queue

3.1 Problem Description

Snapchat's core feature is ephemeral messaging—messages that automatically disappear after viewing or after a time limit. This tests understanding of time-based data structures and queue management.

Requirements:

- Messages have TTL (time-to-live)
- Auto-delete after viewing (Snap behavior)
- Auto-delete after expiration
- Efficient cleanup of expired messages

3.2 Solution

```python
import time
import heapq
from typing import List, Dict, Optional
from collections import defaultdict

class EphemeralMessageQueue:
    def __init__(self):
        self.messages = defaultdict(list)  # recipient -> [messages]
        self.message_map = {}  # message_id -> message data
        self.expiration_heap = []  # (expires_at, message_id)
        self.next_id = 1

    def send_message(self, sender: str, recipient: str,
                     content: str, ttl_seconds: int) -> int:
        """Send message with TTL."""
        message_id = self.next_id
        self.next_id += 1

        expires_at = time.time() + ttl_seconds
        message = {
            'id': message_id,
            'sender': sender,
            'recipient': recipient,
            'content': content,
            'expires_at': expires_at,
            'viewed': False
        }

        self.messages[recipient].append(message)
        self.message_map[message_id] = message
        heapq.heappush(self.expiration_heap, (expires_at, message_id))

        return message_id

    def get_messages(self, recipient: str) -> List[Dict]:
        """Get all non-expired, unviewed messages."""
        self._cleanup_expired()

        current_time = time.time()
        valid_messages = []

        for msg in self.messages[recipient]:
            if (not msg['viewed'] and
                msg['expires_at'] > current_time):
                valid_messages.append(msg)

        return valid_messages
```

```python

    def mark_viewed(self, recipient: str, message_id: int) -> None:
        """Mark as viewed and delete (Snapchat behavior)."""
        if message_id in self.message_map:
            msg = self.message_map[message_id]
            msg['viewed'] = True
            # Remove from recipient's list
            self.messages[recipient] = [
                m for m in self.messages[recipient]
                if m['id'] != message_id
            ]
            del self.message_map[message_id]

    def _cleanup_expired(self) -> None:
        """Remove expired messages."""
        current_time = time.time()

        while self.expiration_heap:
            expires_at, msg_id = self.expiration_heap[0]
            if expires_at > current_time:
                break

            heapq.heappop(self.expiration_heap)
            if msg_id in self.message_map:
                msg = self.message_map[msg_id]
                recipient = msg['recipient']
                self.messages[recipient] = [
                    m for m in self.messages[recipient]
                    if m['id'] != msg_id
                ]
                del self.message_map[msg_id]
```

3.3 Complexity Analysis

- **send_message**: $O(\log n)$ for heap push
- **get_messages**: $O(n + k \log k)$ where k is expired messages
- **mark_viewed**: $O(m)$ where m is messages for recipient
- **Space**: $O(n)$ for storing n messages

4 Problem 5: LRU Cache

4.1 Problem Description

SNAP frequently asks this problem for understanding caching mechanisms in their infrastructure. Must implement get() and put() operations in $O(1)$ time.

4.2 Solution

```python
class ListNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}  # key -> ListNode
        # Dummy head and tail
        self.head = ListNode()
        self.tail = ListNode()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node: ListNode) -> None:
        """Remove node from linked list."""
        node.prev.next = node.next
        node.next.prev = node.prev

    def _add_to_head(self, node: ListNode) -> None:
        """Add node right after head (most recently used)."""
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1

        node = self.cache[key]
        # Move to head (most recently used)
        self._remove(node)
        self._add_to_head(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            # Update existing
            node = self.cache[key]
            node.value = value
            self._remove(node)
            self._add_to_head(node)
        else:
            # Add new
            if len(self.cache) >= self.capacity:
                # Remove LRU (before tail)
                lru = self.tail.prev
                self._remove(lru)
                del self.cache[lru.key]

            new_node = ListNode(key, value)
            self.cache[key] = new_node
```

```python
                self._add_to_head(new_node)
```

4.3 Complexity Analysis

- **get**: $O(1)$ - hash lookup + linked list operations
- **put**: $O(1)$ - hash operations + linked list operations
- **Space**: $O(capacity)$

5 System Design: Snapchat Stories

5.1 Requirements

Design the Stories feature:

- Users can post photo/video Stories (24-hour lifespan)
- Friends can view Stories in chronological order
- Stories auto-delete after 24 hours
- Support 400M+ DAU with low latency
- Handle high upload/view traffic during peak hours

5.2 High-Level Architecture

Components:

1. **Upload Service**: Handle media uploads, compression
2. **Storage Layer**: S3 for media, Cassandra for metadata
3. **CDN**: Cloudflare/Akamai for global distribution
4. **Story Service**: Manage Story creation, retrieval, deletion
5. **TTL Service**: Background job to delete expired Stories
6. **Feed Service**: Generate personalized Story feeds

Data Model (Cassandra):

```
Story {
    story_id: UUID
    user_id: UUID
    media_url: String
    created_at: Timestamp
    expires_at: Timestamp  # created_at + 24h
    view_count: Int
    thumbnail_url: String
}

StoryView {
    story_id: UUID
    viewer_id: UUID
    viewed_at: Timestamp
}
```

Flow:

1. User uploads photo/video → Upload Service
2. Compress/process media → FFmpeg workers
3. Store in S3, create Story record in Cassandra
4. Push to CDN for distribution
5. Notify friends via WebSocket/FCM
6. TTL service periodically scans for expired Stories
7. After 24h: Delete from S3 + Cassandra

Optimizations:

- **CDN Caching**: Cache popular Stories near users
- **Adaptive Bitrate**: Serve different qualities based on bandwidth
- **Preloading**: Prefetch friend Stories in background
- **Sharding**: Partition by user_id for Cassandra

6 Interview Tips and Resources

6.1 Key Success Factors

1. Speed Matters

- SNAP explicitly values fast problem-solving
- Practice solving medium problems in 20-25 minutes
- Have templates ready for common patterns (BFS, Dijkstra, etc.)

2. Runnable Code Required

- No pseudocode—must compile and run
- Test with examples during interview
- Handle edge cases (null, empty, single element)

3. Product Knowledge

- Use Snapchat and understand features deeply
- Know: Stories, Spotlight, Lenses, Chat, Discover, Snap Map
- Discuss trade-offs in context of their products

6.2 Study Resources

Coding Practice:

- LeetCode: Company tag "Snapchat" or "SNAP" (Premium)
- Focus: Graphs (BFS/DFS), Trees, Linked Lists, Arrays
- Difficulty: Medium (70%), Hard (30%)

System Design:

- SNAP Engineering Blog: eng.snap.com
- "Designing Data-Intensive Applications" (Kleppmann)
- Focus: CDN, real-time systems, ephemeral content

Interview Experiences:

- Glassdoor SNAP reviews
- (1Point3Acres) - Chinese forum
- Prepfully SNAP interview guide
- LeetCode Discuss: Snapchat interview threads

6.3 Common Pitfalls

1. **Slow Coding**: SNAP values speed—practice timed coding
2. **Not Testing**: Run through examples with your code
3. **Ignoring Edge Cases**: Null inputs, empty arrays, single elements
4. **Poor Communication**: Explain your approach before coding
5. **Lack of Product Knowledge**: Study Snapchat features

6.4 Behavioral Preparation

SNAP explicitly evaluates three values:

- **Kind**: Collaborative, respectful, inclusive
- **Smart**: Problem-solving, technical depth, learning mindset
- **Creative**: Innovation, bold ideas, thinking differently

Prepare STAR stories demonstrating:

- Times you demonstrated kindness in team settings
- Complex technical problems you solved creatively
- Learning from failures
- Why SNAP/camera/AR technology excites you

7 Conclusion

SNAP interviews test strong algorithmic skills, system design thinking, and alignment with their values. Focus on:

1. **Speed and Accuracy**: Practice timed coding
2. **Graph Algorithms**: Very common at SNAP
3. **Clean Code**: Must be production-ready and runnable
4. **Real-time Systems**: Understand WebSockets, CDN, caching
5. **Product Passion**: Show genuine interest in camera/AR/ephemeral content

Good luck with your SNAP interview!