Topic Introduction

Today's PrepLetter dives into the world of **system design for social media and communication platforms**. We'll be exploring how to build scalable, efficient systems that allow people to interact, share messages, and manage digital content — the backbone of products like Twitter, chat apps, and URL shorteners.

The core concept uniting today's problems is **object-oriented system design with custom data structures**. In interviews, these questions test your ability to model real-world systems, design APIs (functions/classes), and efficiently handle data relationships between users and content.

What does this mean?
Instead of just crunching numbers, you're building virtual "machines" that support operations such as following users, sending messages, and mapping URLs. This requires thinking about:

- **Classes**: How do you represent users, messages, tweets, or URLs?
- **Relationships**: How do users interact? How do messages flow between them?
- **Efficient Lookups**: How do you quickly retrieve a user's feed, chat history, or decode a URL?

A Simple Example:
Imagine designing a basic address book:

- You'd have a `Contact` class (name, phone, email).
- A `ContactList` class might store all contacts and offer add, search, or remove operations.
- You'd want to find a contact quickly, so you might use a dictionary (hash map) for fast lookup by name.

These same ideas scale up to complex systems. In interviews, you're expected to sketch out these designs, often in code, and explain your choices.

Why Group These Problems?

Today's problems all revolve around **social media and communication system designs**:

- **Design Twitter** — Simulates user feeds and following.
- **Design Chat System** — Handles messaging between users.
- **Design URL Shortener** — Maps long URLs to short codes and back.

Each requires custom data structures, smart relationships between objects, and efficient data access patterns. Mastering these helps you build and reason about real-world systems.

Let's jump in!

Problem 1: Design Twitter

Leetcode 355: Design Twitter

Problem Statement (Rephrased):

Design a mini Twitter where users can:

- Post tweets.
- Follow and unfollow other users.
- Get the 10 most recent tweet IDs in their news feed, which includes their own tweets and those of users they follow. Tweets should be ordered from most recent to oldest.

Example:

Suppose:

- User 1 posts tweet 5.
- User 1's feed: [5]
- User 1 follows user 2.
- User 2 posts tweet 6.
- User 1's feed: [6, 5] (most recent first)
- User 1 unfollows user 2.
- User 1's feed: [5]

Try this input:
User 2 posts tweet 7. What is user 1's feed now?

Brute Force Approach:

For each feed request, collect all tweets by the user and everyone they follow, merge them, sort by time, and pick the latest 10.

- **Time Complexity:** O(N log N) per feed request (N = total tweets of user + follows)
- **Drawback:** Slow, especially as tweet counts grow.

Optimal Approach:

- **Core Pattern:**
  - Use classes for users and tweets.
  - Keep each user's tweets in a linked list (or array) with timestamps.
  - Store follow relationships in a dictionary of sets.
  - Use a min-heap (priority queue) to merge tweets efficiently.

Step-by-Step:

- **Tweet Representation:**
  - Store each tweet as (timestamp, tweetId).
- **User Data:**
  - Each user has:
    - A set of followed user IDs (including themselves).
    - A list of their tweets.
- **Posting a Tweet:**
  - Append (timestamp, tweetId) to user's tweet list.
- **Getting News Feed:**
  - For each followed user:

    - Grab their latest tweets.
  - Use a heap to efficiently merge and get the 10 most recent.
- **Follow/Unfollow:**
  - Update the follower's set of followed user IDs.

Python Solution:

```
import heapq
from collections import defaultdict

class Twitter:
    def __init__(self):
        self.timestamp = 0  # Global timestamp for ordering tweets
        self.tweets = defaultdict(list)  # userId -> list of (timestamp, tweetId)
        self.follows = defaultdict(set)  # userId -> set of followed userIds

    def postTweet(self, userId: int, tweetId: int) -> None:
        # Ensure user follows themselves
        self.follows[userId].add(userId)
        self.tweets[userId].append((self.timestamp, tweetId))
        self.timestamp += 1

    def getNewsFeed(self, userId: int) -> list:
        # Gather latest tweets from self and followed users
        heap = []
        for uid in self.follows[userId] | {userId}:
            for t in self.tweets[uid][-10:]:  # Only need last 10 per user
                heapq.heappush(heap, t)
                if len(heap) > 10:
                    heapq.heappop(heap)  # Remove oldest if more than 10 tweets
        # Extract tweets, sort by timestamp descending
        return [tweetId for _, tweetId in sorted(heap, reverse=True)]

    def follow(self, followerId: int, followeeId: int) -> None:
        self.follows[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
        if followeeId != followerId:
            self.follows[followerId].discard(followeeId)
```

Time Complexity:

- `postTweet`: O(1)
- `follow`/`unfollow`: O(1)
- `getNewsFeed`: O(F * 10 log 10) (F = number of followed users)

Space Complexity:

O(U + T) (U = users, T = tweets)

Explanation:

- The `tweets` dictionary holds each user's tweets as (timestamp, tweetId).
- The `follows` set keeps track of follow relationships.
- `getNewsFeed` pulls recent tweets, uses a heap for the top 10.
- We ensure users follow themselves for their own tweets.

Trace Example:

- User 1 posts 5 (timestamp 0): feed [5]
- User 1 follows 2; user 2 posts 6 (timestamp 1): feed [6, 5]
- User 1 unfollows 2: feed [5]

Try this test case:

- User 2 posts 8, 9, and 10 in sequence. What does user 1's feed look like after following user 2 again?

Take a moment to sketch out your own solution before reviewing the code. Try adjusting the logic or data structures to see what changes!

Problem 2: Design Chat System

Design Chat System (hypothetical, but similar to real interview queries)

Problem Statement (Rephrased):

Build a chat system where:

- Users can send messages to other users.
- Users can retrieve their message history with another user, ordered by time.
- Each message contains sender, receiver, content, and timestamp.

Why is this similar?

Like Twitter, you have users, messages, and time ordering. But now, it's about **pairwise communication** and retrieving conversations.

Example:

- User A sends "Hi" to User B.
- User B replies "Hello".
- Chat history between A and B:
  - [("A", "Hi"), ("B", "Hello")]

Try this input:

User A sends "How are you?" to B. What is the chat history now?

Brute Force Approach:

Store all messages in a list. To get chat history between A and B, scan the list and collect messages where (sender, receiver) match.

- **Time Complexity:** O(N) per query (N = total messages)
- **Drawback:** Slow for large histories.

Optimal Approach:

- **Core Pattern:**
  - Use a dictionary to map user pairs to a list of messages.
  - Each message is a tuple (timestamp, sender, content).

Step-by-Step:

- **Message Storage:**
  - For each message, store it under a key like (min(user1, user2), max(user1, user2)) to handle direction.
- **Sending Message:**
  - Append (timestamp, sender, content) to the history list for that user pair.
- **Getting History:**
  - Retrieve and return the list for the user pair.

Pseudocode:

```
Initialize timestamp = 0
Initialize chat_history as dict mapping (userA, userB) to list

Function sendMessage(sender, receiver, content):
    key = (min(sender, receiver), max(sender, receiver))
    Append (timestamp, sender, content) to chat_history[key]
    Increment timestamp

Function getChatHistory(user1, user2):
    key = (min(user1, user2), max(user1, user2))
    Return chat_history[key] (sorted by timestamp, already in order)
```

Time Complexity:

- `sendMessage`: O(1)
- `getChatHistory`: O(M) (M = number of messages between those two users)

Try this test case:

User C sends "Hey" to D, then D sends "What's up?" to C. What is their chat history?

Trace Example:

Let's walk through:

- sendMessage("A", "B", "Hi") at t=0
- sendMessage("B", "A", "Hello") at t=1
- sendMessage("A", "B", "How are you?") at t=2

History between A and B:
[(0, "A", "Hi"), (1, "B", "Hello"), (2, "A", "How are you?")]

Try implementing this with your own code!

Problem 3: Design URL Shortener

Leetcode 535: Encode and Decode TinyURL

Problem Statement (Rephrased):

Build a system that:

- Converts long URLs to short URLs.
- Allows retrieving the original long URL from a short one.

What's different?
Instead of users and feeds, we're mapping strings (long URLs) to unique short codes. The underlying pattern is still **efficient mapping and retrieval**, like in the previous problems.

Example:

- encode("https://leetcode.com/problems/encode-and-decode-tinyurl/") -> "http://tinyurl.com/abcd1"
- decode("http://tinyurl.com/abcd1") -> "https://leetcode.com/problems/encode-and-decode-tinyurl/"

Try this input:
Encode "https://www.google.com/". What short code might you get?

Brute Force Approach:

Assign incremental IDs for each long URL, map to a short code, and store mapping in a dictionary.

- **Time Complexity:** O(1) for both encode and decode.
- **Drawback:** Codes can become predictable.

Optimal Approach:

- **Core Pattern:**
  - Use a dictionary to store long<->short mappings.
  - Generate short codes (random or incremental).
  - Ensure code uniqueness (check for collisions).

Pseudocode:

```
Initialize long_to_short and short_to_long as dicts
Base URL = "http://tinyurl.com/"
Possible chars = "0-9a-zA-Z"

Function encode(longUrl):
    If longUrl in long_to_short:
        Return Base URL + long_to_short[longUrl]
    Repeat:
        Generate random 6-char code
```

```
    Until code not in short_to_long
    Store mappings in both dicts
    Return Base URL + code

Function decode(shortUrl):
    Extract code from shortUrl
    Return short_to_long[code]
```

Time Complexity:
O(1) for both `encode` and `decode`

Try this test case:
Encode and decode "https://openai.com/".

Reflect:
What if two users try to encode the same URL? Should it return the same code, or a new one every time? Think about design decisions!

Summary and Next Steps

Today, you tackled three classic **system/object design problems**:

- **Why grouped?** They all model real-world social media and communication platforms, focusing on relationships, efficient data access, and practical API design.
- **Key patterns:**
  - Use classes and dictionaries to model users, messages, and content.
  - Efficient retrieval through direct mappings (hash maps).
  - Handling relationships (followers, chat pairs).
  - Generating unique IDs/codes.

Common pitfalls:

- Forgetting to ensure uniqueness (user following themselves, URL code collisions).
- Not optimizing for efficient lookups (scanning all messages instead of using a map).
- Mishandling edge cases (unfollowing oneself, duplicate messages, etc).

Action List

- Solve all three problems yourself, even the one with code provided.
- Try re-implementing Problem 2 and 3 using a different approach, such as using databases, or alternate data structures.
- Explore other system design problems (e.g., design Instagram, Facebook, or collaborative document editors).
- Compare your solutions with others for edge-case handling and code clarity.
- If you get stuck, review the patterns — practice is key!

Keep building, keep questioning, and you'll be ready for even the toughest system design interviews!