TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: **117667**

Session 1: Stacks, Queues, and Two Pointer

Session Overview

In this session we will continue to work with linear data structures like strings and arrays. Students will learn about two new linear data structures: stacks and queues. They will learn new techniques to optimally iterate through these data structures in order to solve problems including validating nesting of parentheses, reversing complex strings, finding the middle of a list, etc.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

🎢 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as "solving the problem" - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
  - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution

- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 1

▾ Standard Problem Set Version 1

Problem 1: Post Format Validator

You are managing a social media platform and need to ensure that posts are properly formatted. Each post must have balanced and correctly nested tags, such as `()` for mentions, `[]` for hashtags, and `{}` for links. You are given a string representing a post's content, and your task is to determine if the tags in the post are correctly formatted.

A post is considered valid if:

1. Every opening tag has a corresponding closing tag of the same type.
2. Tags are closed in the correct order.

```
def is_valid_post_format(posts):
  pass
```

Example Usage:

```python
print(is_valid_post_format("()"))
print(is_valid_post_format("()[]{}"))
print(is_valid_post_format("(]"))
```

Example Output:

```
True
True
False
```

▶ 💡 **Hint: Stacks**

▶ 💡 **Hint: Pseudocode**

Problem 2: Reverse User Comments Queue

On your platform, comments on posts are displayed in the order they are received. However, for a special feature, you need to reverse the order of comments before displaying them. Given a queue of comments represented as a list of strings, reverse the order using a stack.

```python
def reverse_comments_queue(comments):
  pass
```

Example Usage:

```python
print(reverse_comments_queue(["Great post!", "Love it!", "Thanks for sharing."]))

print(reverse_comments_queue(["First!", "Interesting read.", "Well written."]))
```

Example Output:

```
['Thanks for sharing.', 'Love it!', 'Great post!']
['Well written.', 'Interesting read.', 'First!']
```

Problem 3: Check Symmetry in Post Titles

As part of a new feature on your social media platform, you want to highlight post titles that are symmetrical, meaning they read the same forwards and backwards when ignoring spaces, punctuation, and case. Given a post title as a string, use a new algorithmic technique the **two-pointer method** to determine if the title is symmetrical.

```python
def is_symmetrical_title(title):
  pass
```

Example Usage:

```
print(is_symmetrical_title("A Santa at NASA"))
print(is_symmetrical_title("Social Media"))
```

Example Output:

```
True
False
```

▶ 💡 **Hint: Two Pointer Technique**

Problem 4: Engagement Boost

You track your daily engagement rates as a list of integers, sorted in non-decreasing order. To analyze the impact of certain strategies, you decide to square each engagement rate and then sort the results in non-decreasing order.

Given an integer array `engagements` sorted in non-decreasing order, return an array of the squares of each number sorted in non-decreasing order.

Your Task:

- Read through the existing solution and add comments so that everyone in your pod understands how it works.
- Modify the solution below to use the two-pointer technique.

```
def engagement_boost(engagements):
    squared_engagements = []

    for i in range(len(engagements)):
        squared_engagement = engagements[i] * engagements[i]
        squared_engagements.append((squared_engagement, i))

    squared_engagements.sort(reverse=True)

    result = [0] * len(engagements)
    position = len(engagements) - 1

    for square, original_index in squared_engagements:
        result[position] = square
        position -= 1

    return result
```

Example Usage:

```
print(engagement_boost([-4, -1, 0, 3, 10]))
print(engagement_boost([-7, -3, 2, 3, 11]))
```

Example Output:

```
[0, 1, 9, 16, 100]
[4, 9, 9, 49, 121]
```

Problem 5: Content Cleaner

You want to make sure your posts are clean and professional. Given a string `post` of lowercase and uppercase English letters, you want to remove any pairs of adjacent characters where one is the lowercase version of a letter and the other is the uppercase version of the same letter. Keep removing such pairs until the post is clean.

A clean post does not have two adjacent characters `post[i]` and `post[i + 1]` where:

- `post[i]` is a lowercase letter and `post[i + 1]` is the same letter in uppercase or vice-versa.

Return the clean post.

Note that an empty string is also considered clean.

```python
def clean_post(post):
  pass
```

Example Usage:

```python
print(clean_post("poOost"))
print(clean_post("abBAcC"))
print(clean_post("s"))
```

Example Output:

```
post

s
```

▶ 💡 **Hint: Choosing the Right Approach**

▶ 💡 **Hint: Useful Built-In Methods**

Problem 6: Post Editor

You want to add a creative twist to your posts by reversing the order of characters in each word within your post while still preserving whitespace and the initial word order. Given a string `post`, use a queue to reverse the order of characters in each word within the sentence.

```python
def edit_post(post):
  pass
```

Example Usage:

```python
print(edit_post("Boost your engagement with these tips"))
print(edit_post("Check out my latest vlog"))
```

Example Output:

```
tsooB ruoy tnemegegna htiw esehT spit
kcehC tuo ym tseval golv
```

▶ 💡 **Hint: Queues**

Problem 7: Post Compare

You often draft your posts and edit them before publishing. Given two draft strings `draft1` and `draft2`, return `true` if they are equal when both are typed into empty text editors. `'#'` means a backspace character.

Note that after backspacing an empty text, the text will remain empty.

```python
def post_compare(draft1, draft2):
  pass
```

Example Usage:

```python
print(post_compare("ab#c", "ad#c"))
print(post_compare("ab##", "c#d#"))
print(post_compare("a#c", "b"))
```

Example Output:

```
True
True
False
```

▶ 💡 **Hint: Helper Functions**

Close Section

▾ Standard Problem Set Version 2

Problem 1: Time Needed to Stream Movies

There are `n` users in a queue waiting to stream their favorite movies, where the 0th user is at the front of the queue and the `(n - 1)` th user is at the back of the queue.

You are given a 0-indexed integer array `movies` of length `n` where the number of movies that the `i` th user would like to stream is `movies[i]`.

Each user takes exactly 1 second to stream a movie. A user can only stream 1 movie at a time and has to go back to the end of the queue (which happens instantaneously) in order to stream more movies. If a user does not have any movies left to stream, they will leave the queue.

Return the time taken for the user at position `k` (0-indexed) to finish streaming all their movies.

```
def time_required_to_stream(movies, k):
  pass
```

Example Usage:

```
print(time_required_to_stream([2, 3, 2], 2))
print(time_required_to_stream([5, 1, 1, 1], 0))
```

Example Output:

```
6
8
```

▶ 💡 **Hint: Queues**

▶ 💡 **Hint: Pseudocode**

Problem 2: Reverse Watchlist

You are given a list `watchlist` representing a list of shows sorted by popularity for a particular user. The user wants to discover new shows they haven't heard of before by reversing the list to show the least popular shows first.

Using the two-pointer approach, implement a function `reverse_watchlist()` that reverses the order of the `watchlist` in-place. This means that the first show in the given list should become the last, the second show should become the second to last, and so on. Return the reversed list.

Do not use list slicing (e.g., watchlist[::-1]) to achieve this.

```
def reverse_watchlist(watchlist):
  pass
```

Example Usage:

```
watchlist = ["Breaking Bad", "Stranger Things", "The Crown", "The Witcher"]

print(reverse_watchlist(watchlist))
```

Example Output:

```
['The Witcher', 'The Crown', 'Stranger Things', 'Breaking Bad']
```

▶ 💡 **Hint: Two Pointer Technique**

Problem 3: Remove All Adjacent Duplicate Shows

You are given a string `schedule` representing the lineup of shows on a streaming platform, where each character in the string represents a different show. A duplicate removal consists of choosing two adjacent and equal shows and removing them from the schedule.

We repeatedly make duplicate removals on `schedule` until no further removals can be made.

Return the final schedule after all such duplicate removals have been made. The answer is guaranteed to be unique.

```
def remove_duplicate_shows(schedule):
  pass
```

Example Usage:

```
print(remove_duplicate_shows("abbaca"))
print(remove_duplicate_shows("azxxzy"))
```

Example Output:

```
ca
ay
```

▶ 💡 **Hint: Stacks**

▶ 💡 **Hint: Pseudocode**

Problem 4: Minimum Average of Smallest and Largest View Counts

You manage a collection of view counts for different shows on a streaming platform. You are given an array `view_counts` of `n` integers, where `n` is even.

You repeat the following procedure `n / 2` times:

1. Remove the show with the smallest view count, `min_view_count`, and the show with the largest view count, `max_view_count`, from `view_counts`.
2. Add `(min_view_count + max_view_count) / 2` to the list of average view counts `average_views`.

Return the minimum element in `average_views`.

```
def minimum_average_view_count(view_counts):
  pass
```

Example Usage:

```
print(minimum_average_view_count([7, 8, 3, 4, 15, 13, 4, 1]))
print(minimum_average_view_count([1, 9, 8, 3, 10, 5]))
print(minimum_average_view_count([1, 2, 3, 7, 8, 9]))
```

Example Output:

```
5.5
5.5
5.0
```

Problem 5: Minimum Remaining Watchlist After Removing Movies

You have a watchlist consisting only of uppercase English letters representing movies. Each movie is represented by a unique letter.

You can apply some operations to this watchlist where, in one operation, you can remove any occurrence of one of the movie pairs "AB" or "CD" from the watchlist.

Return the minimum possible length of the modified watchlist that you can obtain.

Note that the watchlist concatenates after removing the movie pair and could produce new "AB" or "CD" pairs.

```
def min_remaining_watchlist(watchlist):
  pass
```

Example Usage:

```
print(min_remaining_watchlist("ABFCACDB"))
print(min_remaining_watchlist("ACBBD"))
```

Example Output:

```
2
5
```

Problem 6: Apply Operations to Show Ratings

You are given a 0-indexed array `ratings` of size `n` consisting of non-negative integers. Each integer represents the rating of a show in a streaming service.

You need to apply `n - 1` operations to this array where, in the `i`th operation (0-indexed), you will apply the following on the `i`th element of ratings:

- If `ratings[i] == ratings[i + 1]`, then multiply `ratings[i]` by 2 and set `ratings[i + 1]` to 0. Otherwise, you skip this operation.

After performing all the operations, shift all the 0's to the end of the array.

For example, the array `[1,0,2,0,0,1]` after shifting all its 0's to the end, is `[1,2,1,0,0,0]`.

Return the resulting array of ratings.

```
def apply_rating_operations(ratings):
  pass
```

Example Usage:

```
print(apply_rating_operations([1, 2, 2, 1, 1, 0]))
print(apply_rating_operations([0, 1]))
```

Example Output:

```
[1, 4, 2, 0, 0, 0]
[1, 0]
```

Problem 7: Lexicographically Smallest Watchlist

You are managing a watchlist for a streaming service, represented by a string `watchlist` consisting of lowercase English letters, where each letter represents a different show.

You are allowed to perform operations on this watchlist. In one operation, you can replace a show in `watchlist` with another show (i.e., another lowercase English letter).

Your task is to make the watchlist a palindrome with the minimum number of operations possible. If there are multiple palindromes that can be made using the minimum number of operations, make the lexicographically smallest one.

A string `a` is lexicographically smaller than a string `b` (of the same length) if in the first position where `a` and `b` differ, string `a` has a letter that appears earlier in the alphabet than the corresponding letter in `b`.

Return the resulting watchlist string.

Implement the following pseudocode:

```
1. Convert the watchlist string to a list.
2. Initialize two pointers:
   * Left Pointer: Start at the beginning of the list (index 0).
   * Right Pointer: Start at the end of the list (last index).
3. While the left pointer is less than the right pointer:
   a. Compare the characters at the left and right pointers.
   b. If the characters are different:
      * Replace the character that is alphabetically later (greater) with the one th
   c. Move the left pointer one step to the right.
   d. Move the right pointer one step to the left.
4. Convert the list back to a string.
5. Return the resulting string.
```

```python
def make_smallest_watchlist(watchlist):
  pass
```

Example Usage:

```python
print(make_smallest_watchlist("egcfe"))
print(make_smallest_watchlist("abcd"))
print(make_smallest_watchlist("seven"))
```

Example Output:

```
efcfe
abba
neven
```

Close Section

▸ Advanced Problem Set Version 1

▸ Advanced Problem Set Version 2