TIP102 | Intermediate Technical Interview Prep

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

> 👋 **IMPORTANT:** This session is fully **asynchronous**. Please use these questions to practice further!

Session 2: Review

Session Overview

In this unit, we will transition from the UPI method to the full UMPIRE method. Students will review content from Units 1-3 by matching each problem to a data structure and/or strategy introduced in previous units before solving. Students will also practice evaluating the time and space complexity of each of problem. Problems will cover strings, arrays, hash tables (dictionaries), stacks, queues, and the two pointer technique.

> 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 2

▼ Standard Problem Set Version 1

Problem 1: Planning Your Daily Work Schedule

Your day consists of various tasks, each requiring a certain amount of time. To optimize your workday, you want to find a pair of tasks that fits exactly into a specific time slot you have available. You need to identify if there is a pair of tasks whose combined time matches the available slot.

Given a list of integers representing the time required for each task and an integer representing the available time slot, write a function that returns `True` if there exists a pair of tasks that exactly matches the available time slot, and `False` otherwise.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def find_task_pair(task_times, available_time):
  pass
```

Example Usage:

```
task_times = [30, 45, 60, 90, 120]
available_time = 105
print(find_task_pair(task_times, available_time))

task_times_2 = [15, 25, 35, 45, 55]
available_time = 100
print(find_task_pair(task_times_2, available_time))

task_times_3 = [20, 30, 50, 70]
available_time = 60
print(find_task_pair(task_times_3, available_time))
```

Example Output:

```
True
True
False
```

Problem 2: Minimizing Workload Gaps

You work with clients across different time zones and often have gaps between your work sessions. You want to minimize these gaps to make your workday more efficient. You have a list of work sessions, each with a start time and an end time. Your task is to find the smallest gap between any two consecutive work sessions.

Given a list of tuples where each tuple represents a work session with a start and end time (both in 24-hour format as integers, e.g., 1300 for 1:00 PM), write a function to find the smallest gap between any two consecutive work sessions. The gap is measured in minutes.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def find_smallest_gap(work_sessions):
  pass
```

Example Usage:

```
work_sessions = [(900, 1100), (1300, 1500), (1600, 1800)]
print(find_smallest_gap(work_sessions))

work_sessions_2 = [(1000, 1130), (1200, 1300), (1400, 1500)]
print(find_smallest_gap(work_sessions_2))

work_sessions_3 = [(900, 1100), (1115, 1300), (1315, 1500)]
print(find_smallest_gap(work_sessions_3))
```

Example Output:

```
60
30
15
```

Problem 3: Expense Tacking and Categorization

You travel frequently and need to keep track of your expenses. You categorize your expenses into different categories such as "Food," "Transport," "Accommodation," etc. At the end of each month, you want to calculate the total expenses for each category to better understand where your money is going.

Given a list of tuples where each tuple contains an expense category (string) and an expense amount (float), write a function that returns the expense categories and the total expenses for each category. Additionally, the function should return the category with the highest total expense.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def calculate_expenses(expenses):
  pass
```

Example Usage:

```
expenses = [("Food", 12.5), ("Transport", 15.0), ("Accommodation", 50.0),
            ("Food", 7.5), ("Transport", 10.0), ("Food", 10.0)]
print(calculate_expenses(expenses))

expenses_2 = [("Entertainment", 20.0), ("Food", 15.0), ("Transport", 10.0),
              ("Entertainment", 5.0), ("Food", 25.0), ("Accommodation", 40.0)]
print(calculate_expenses(expenses_2))

expenses_3 = [("Utilities", 100.0), ("Food", 50.0), ("Transport", 75.0),
              ("Utilities", 50.0), ("Food", 25.0)]
print(calculate_expenses(expenses_3))
```

Example Output:

```
({'Food': 30.0, 'Transport': 25.0, 'Accommodation': 50.0}, 'Accommodation')
({'Entertainment': 25.0, 'Food': 40.0, 'Transport': 10.0, 'Accommodation': 40.0}, 'F
({'Utilities': 150.0, 'Food': 75.0, 'Transport': 75.0}, 'Utilities')
```

Problem 4: Analyzing Word Frequency

As a digital nomad who writes blogs, articles, and reports regularly, it's important to analyze the text you produce to ensure clarity and avoid overusing certain words. You want to create a tool that analyzes the frequency of each word in a given text and identifies the most frequent word(s).

Given a string of text, write a function that returns the unique words and the number of times each word appears in the text. Additionally, return a list of the word(s) that appear most frequently.

Assumptions:

- The text is case-insensitive, so `"Word"` and `"word"` should be treated as the same word.
- Punctuation should be ignored.
- In case of a tie, return all words that have the highest frequency.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def word_frequency_analysis(text):
  pass
```

Example Usage:

```
text = "The quick brown fox jumps over the lazy dog. The dog was not amused."
print(word_frequency_analysis(text))

text_2 = "Digital nomads love to travel. Travel is their passion."
print(word_frequency_analysis(text_2))

text_3 = "Stay connected. Stay productive. Stay happy."
print(word_frequency_analysis(text_3))
```

Example Output:

```
({'the': 3, 'quick': 1, 'brown': 1, 'fox': 1, 'jumps': 1, 'over': 1, 'lazy': 1, 'dog
({'digital': 1, 'nomads': 1, 'love': 1, 'to': 1, 'travel': 2, 'is': 1, 'their': 1, '
({'stay': 3, 'connected': 1, 'productive': 1, 'happy': 1}, ['stay'])
```

Problem 5: Validating HTML Tags

As a digital nomad who frequently writes and edits HTML for your blog, you want to ensure that your HTML code is properly structured. One important aspect of HTML structure is ensuring that all opening tags have corresponding closing tags and that they are properly nested.

Given a string of HTML-like tags (simplified for this problem), write a function to determine if the tags are properly nested and closed. The tags will be in the form of `<tag>` for opening tags and `</tag>` for closing tags.

The function should return `True` if the tags are properly nested and closed, and `False` otherwise.

Assumptions:

- You can assume that tags are well-formed (e.g., `<div>`, `</div>`, `<a>`, `</a>`, etc.).
- Tags can be nested but cannot overlap improperly (e.g., `<div><p></div></p>` is invalid).

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def validate_html_tags(html):
  pass
```

Example Usage:

```
html = "<div><p></p></div>"
print(validate_html_tags(html))

html_2 = "<div><p></div></p>"
print(validate_html_tags(html_2))

html_3 = "<div><p><a></a></p></div>"
print(validate_html_tags(html_3))

html_4 = "<div><p></a></p></div>"
print(validate_html_tags(html_4))
```

Example Output:

```
True
False
True
False
```

Problem 6: Task Prioritization with Limited Time

You often have a long list of tasks to complete, but limited time to do so. Each task has a specific duration, and you only have a certain amount of time available in your schedule. You need to prioritize and complete as many tasks as possible within the given time limit.

Given a list of task durations and a time limit, determine the maximum number of tasks you can complete within that time.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def max_tasks_within_time(tasks, time_limit):
  pass
```

Example Usage:

```
tasks = [5, 10, 7, 8]
time_limit = 20
print(max_tasks_within_time(tasks, time_limit))

tasks_2 = [2, 4, 6, 3, 1]
time_limit = 10
print(max_tasks_within_time(tasks_2, time_limit))

tasks_3 = [8, 5, 3, 2, 7]
time_limit = 15
print(max_tasks_within_time(tasks_3, time_limit))
```

Example Output:

```
3
4
3
```

Problem 7: Frequent Co-working Spaces

You often work from various co-working spaces. You want to analyze your usage patterns to identify which co-working spaces you visit the most frequently. Given a list of co-working spaces you visited over the past month, write a function to determine which co-working space(s) you visited most frequently. If there is a tie, return all of the most visited spaces.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def most_frequent_spaces(visits):
  pass
```

Example Usage:

```
visits = ["WeWork", "Regus", "Spaces", "WeWork", "Regus", "WeWork"]
print(most_frequent_spaces(visits))

visits_2 = ["IndieDesk", "Spaces", "IndieDesk", "WeWork", "Spaces", "IndieDesk", "WeW
print(most_frequent_spaces(visits_2))

visits_3 = ["Hub", "Regus", "WeWork", "Hub", "WeWork", "Regus", "Hub", "Regus"]
print(most_frequent_spaces(visits_3))
```

Example Output:

```
['WeWork']
['IndieDesk']
['Hub', 'Regus']
```

Problem 8: Track Popular Destinations

You want to track the most popular destinations you visited based on the number of times you have visited them. Given a list of visited destinations with timestamps, your goal is to determine the destination that has been visited the most and the total number of times it was visited. If there is a tie, return the one with the latest visit.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def most_popular_destination(visits):
  pass
```

Example Usage:

```
visits = [("Paris", "2024-07-15"), ("Tokyo", "2024-08-01"), ("Paris", "2024-08-05"),
print(most_popular_destination(visits))

visits_2 = [("London", "2024-06-01"), ("Berlin", "2024-06-15"), ("London", "2024-07-
print(most_popular_destination(visits_2))

visits_3 = [("Sydney", "2024-05-01"), ("Dubai", "2024-05-15"), ("Sydney", "2024-05-2
print(most_popular_destination(visits_3))
```

Example Output:

```
('Paris', 3)
('London', 3)
('Dubai', 3)
```

Close Section

▾ Standard Problem Set Version 2

Problem 1: Track Podcast Episodes by Length

You are managing a podcast and need to analyze the lengths of the episodes. Given a list of episodes where each episode is represented by its duration in minutes, you want to determine how many episodes fall into each of the following time ranges: less than 30 minutes, 30 to 60 minutes, and more than 60 minutes.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def track_episode_lengths(episode_lengths):
  pass
```

Example Usage:

```
episode_lengths = [15, 45, 32, 67, 22, 59, 70]
print(track_episode_lengths(episode_lengths))

episode_lengths_2 = [10, 25, 30, 45, 55, 65, 80]
print(track_episode_lengths(episode_lengths_2))

episode_lengths_3 = [30, 30, 30, 30, 30]
print(track_episode_lengths(episode_lengths_3))
```

Example Output:

```
(2, 3, 2)
(2, 3, 2)
(0, 5, 0)
```

Problem 2: Identify Longest Episode

Given a list of episode durations from a podcast series, your task is to identify the longest episode. If there are multiple episodes with the maximum duration, return the duration of the longest episode.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def identify_longest_episode(durations):
  pass
```

Example Usage:

```
print(identify_longest_episode([30, 45, 60, 45, 30]))
print(identify_longest_episode([20, 30, 40, 40, 30, 20]))
print(identify_longest_episode([55, 60, 55, 60, 60]))
```

Example Output:

```
60
40
60
```

Problem 3: Find Most Frequent Episode Length

You are given a list of episode lengths from a podcast series. Your task is to determine which episode length occurs most frequently. If there are multiple lengths with the same highest frequency, return the smallest episode length.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def most_frequent_length(episode_lengths):
  pass
```

Example Usage:

```
print(most_frequent_length([30, 45, 30, 60, 45, 30]))
print(most_frequent_length([20, 20, 30, 30, 40, 40, 40]))
print(most_frequent_length([50, 60, 70, 80, 90, 100]))
```

Example Output:

```
30
40
50
```

Problem 4: Find Median Episode Length

Given a list of episode durations from a podcast series, find the median episode length. The median is the middle value when the list is sorted. If the list has an even number of elements, return the average of the two middle values.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def find_median_episode_length(durations):
  pass
```

Example Usage:

```
print(find_median_episode_length([45, 30, 60, 30, 90]))
print(find_median_episode_length([90, 80, 60, 70, 50]))
print(find_median_episode_length([30, 10, 20, 40, 30, 50]))
```

Example Output:

```
45
70
30.0
```

Problem 5: Find Unique Genres with Minimum Episode Length

Given a list of podcast episodes, each with a genre and length, find the unique genres where the shortest episode length is greater than or equal to a specified threshold. Return a list of these genres sorted alphabetically.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def unique_genres_with_min_length(episodes, threshold):
  pass
```

Example Usage:

```
print(unique_genres_with_min_length([("Episode 1", "Tech", 30), ("Episode 2", "Healt
print(unique_genres_with_min_length([("Episode A", "Science", 40), ("Episode B", "Sc
print(unique_genres_with_min_length([("Episode X", "Music", 20), ("Episode Y", "Musi
```

Example Output:

```
['Entertainment', 'Health', 'Tech']
['Art', 'Science']
['Drama', 'Music']
```

Problem 6: Find Recent Podcast Episodes

You are developing a podcast management system and need to keep track of the most recent podcast episodes. Given a list of episodes where each episode is represented by a unique ID, you need to implement a function that retrieves the most recent `n` episodes from the list in the order they were added.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def get_recent_episodes(episodes, n):
  pass
```

Example Usage:

```
episodes1 = ['episode1', 'episode2', 'episode3', 'episode4']
n = 3
print(get_recent_episodes(episodes1, n))

episodes2 = ['ep1', 'ep2', 'ep3']
n = 2
print(get_recent_episodes(episodes2, n))

episodes3 = ['a', 'b', 'c', 'd']
n = 5
print(get_recent_episodes(episodes3, n))
```

Example Output:

```
['episode4', 'episode3', 'episode2']
['ep3', 'ep2']
['d', 'c', 'b', 'a']
```

Problem 7: Reorder Podcast Episodes

You are designing a feature for a podcast app that allows users to reorder their list of episodes. The episodes are initially in a stack (LIFO order). Write a function to reorder the episodes based on a list of indices specifying the new order. The indices are 0-based and represent the new position of each episode in the stack.

For instance, if the stack contains episodes `[A, B, C, D]` and the indices are `[2, 0, 3, 1]`, it means that the episode originally at index `0` should move to index `2`, the episode at index `1` should move to index `0`, and so on.

The function should return the reordered list of episodes.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def reorder_stack(stack, indices):
  pass
```

Example Usage:

```
stack1 = ['Episode1', 'Episode2', 'Episode3', 'Episode4']
indices = [2, 0, 3, 1]
print(reorder_stack(stack1, indices))

stack2 = ['A', 'B', 'C', 'D']
indices = [1, 2, 3, 0]
print(reorder_stack(stack2, indices))

stack3 = ['Alpha', 'Beta', 'Gamma']
indices = [0, 2, 1]
print(reorder_stack(stack3, indices))
```

Example Output:

```
['Episode2', 'Episode4', 'Episode1', 'Episode3']
['D', 'A', 'B', 'C']
['Alpha', 'Gamma', 'Beta']
```

Problem 8: Find Longest Consecutive Listen Gaps

You are building a feature for a podcast app that helps users identify the longest period of time between listening to consecutive episodes of a podcast. Given a list of episode listen timestamps (in minutes since midnight) sorted in ascending order, your task is to determine the longest gap between consecutive listens.

Write a function to find the longest gap between consecutive listens.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def find_longest_gap(timestamps):
  pass
```

Example Usage:

```
timestamps1 = [30, 50, 70, 100, 120, 150]
print(find_longest_gap(timestamps1))

timestamps2 = [10, 20, 30, 50, 60, 90]
print(find_longest_gap(timestamps2))

timestamps3 = [5, 10, 15, 25, 35, 45]
print(find_longest_gap(timestamps3))
```

Example Output:

```
30
30
10
```

Close Section

▸ Advanced Problem Set Version 1

▸ Advanced Problem Set Version 2