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**
▶ **Standard Problem Set Version 2**
▼ **Advanced Problem Set Version 1**

Problem 1: Count Unique Characters in a Script

Given a dictionary where the keys are character names and the values are lists of their dialogue lines, count the number of unique characters in the script.

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 count_unique_characters(script):
  pass
```

Example Usage:

```
script = {
    "Alice": ["Hello there!", "How are you?"],
    "Bob": ["Hi Alice!", "I'm good, thanks!"],
    "Charlie": ["What's up?"]
}
print(count_unique_characters(script))

script_with_redundant_keys = {
    "Alice": ["Hello there!"],
    "Alice": ["How are you?"],
    "Bob": ["Hi Alice!"]
}
print(count_unique_characters(script_with_redundant_keys))
```

Example Output:

```
3
2
```

Problem 2: Find Most Frequent Keywords

Identify the most frequently used keywords from a dictionary where the keys are scene names and the values are lists of keywords used in each scene. Return the keyword that appears the most frequently across all scenes. If there is a tie, return all the keywords with 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 find_most_frequent_keywords(scenes):
  pass
```

Example Usage:

```
scenes = {
    "Scene 1": ["action", "hero", "battle"],
    "Scene 2": ["hero", "action", "quest"],
    "Scene 3": ["battle", "strategy", "hero"],
    "Scene 4": ["action", "strategy"]
}
print(find_most_frequent_keywords(scenes))

scenes = {
    "Scene A": ["love", "drama"],
    "Scene B": ["drama", "love"],
    "Scene C": ["comedy", "love"],
    "Scene D": ["comedy", "drama"]
}
print(find_most_frequent_keywords(scenes))
```

Example Output:

```
['action', 'hero']
['love', 'drama']
```

Problem 3: Track Scene Transitions

Given a list of scenes in a story, use a queue to keep track of the transitions from one scene to the next. You need to simulate the transitions by processing each scene in the order they appear and print out each transition from the current scene to the next.

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_scene_transitions(scenes):
  pass
```

Example Usage:

```
scenes = ["Opening", "Rising Action", "Climax", "Falling Action", "Resolution"]
track_scene_transitions(scenes)

scenes = ["Introduction", "Conflict", "Climax", "Denouement"]
track_scene_transitions(scenes)
```

Example Output:

```
Transition from Opening to Rising Action
Transition from Rising Action to Climax
Transition from Climax to Falling Action
Transition from Falling Action to Resolution

Transition from Introduction to Conflict
Transition from Conflict to Climax
Transition from Climax to Denouement
```

Problem 4: Organize Scene Data by Date

Given a list of scene records, where each record contains a date and a description, sort the list by date and return the sorted list. Each record is a tuple where the first element is the date in `YYYY-MM-DD` format and the second element is the description of the scene.

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.

```python
def organize_scene_data_by_date(scene_records):
  pass
```

Example Usage:

```python
scene_records = [
    ("2024-08-15", "Climax"),
    ("2024-08-10", "Introduction"),
    ("2024-08-20", "Resolution"),
    ("2024-08-12", "Rising Action")
]
print(organize_scene_data_by_date(scene_records))

scene_records = [
    ("2023-07-05", "Opening"),
    ("2023-07-07", "Conflict"),
    ("2023-07-01", "Setup"),
    ("2023-07-10", "Climax")
]
print(organize_scene_data_by_date(scene_records))
```

Example Output:

```
[('2024-08-10', 'Introduction'), ('2024-08-12', 'Rising Action'), ('2024-08-15', 'Cl
[('2023-07-01', 'Setup'), ('2023-07-05', 'Opening'), ('2023-07-07', 'Conflict'), ('2(
```

Problem 5: Filter Scenes by Keyword

Scenes often contain descriptions that set the tone or provide important information. However, certain scenes may need to be filtered out based on keywords that are either irrelevant to the current narrative path or that the user wishes to avoid. Write a function that, given a list of scene descriptions and a keyword, filters out the scenes that contain the specified keyword.

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.

```python
def filter_scenes_by_keyword(scenes, keyword):
  pass
```

Example Usage:

```python
scenes = [
    "The hero enters the dark forest.",
    "A mysterious figure appears.",
    "The hero finds a hidden treasure.",
    "An eerie silence fills the air."
]
keyword = "hero"

filtered_scenes = filter_scenes_by_keyword(scenes, keyword)
print(filtered_scenes)

scenes = [
    "The spaceship lands on an alien planet.",
    "A strange creature approaches the crew.",
    "The crew prepares to explore the new world."
]
keyword = "crew"

filtered_scenes = filter_scenes_by_keyword(scenes, keyword)
print(filtered_scenes)
```

Example Output:

```
['An eerie silence fills the air.', 'A mysterious figure appears.']
['The spaceship lands on an alien planet.']
```

Problem 6: Manage Character Arcs

Character arcs are crucial to maintaining a coherent narrative. These arcs often involve a series of events or changes that must occur in a specific order. As the story progresses, you may need to add, remove, or update these events to ensure the character's development follows the intended sequence.

Your task is to simulate managing character arcs using a stack. Given a series of events representing a character's development, use a stack to process these events. Add events to the stack as they occur and pop them off when they are completed or no longer relevant, ensuring that the character arc maintains the correct sequence.

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 manage_character_arc(events):
  pass
```

Example Usage:

```
events = [
    "Character is introduced.",
    "Character faces a dilemma.",
    "Character makes a decision.",
    "Character grows stronger.",
    "Character achieves goal."
]

processed_arc = manage_character_arc(events)
print(processed_arc)

events = [
    "Character enters a new world.",
    "Character struggles to adapt.",
    "Character finds a mentor.",
    "Character gains new skills.",
    "Character faces a major setback.",
    "Character overcomes the setback."
]

processed_arc = manage_character_arc(events)
print(processed_arc)
```

Example Output:

```
['Character is introduced.', 'Character faces a dilemma.', 'Character makes a decisi
['Character enters a new world.', 'Character struggles to adapt.', 'Character finds
```

Problem 7: Identify Repeated Themes

Themes often recur across different scenes to reinforce key ideas or emotions. Identifying these repeated themes is crucial for analyzing the narrative structure and ensuring thematic consistency. Write a function that, given a list of scenes with their associated themes, identifies themes that appear more than once and returns a list of these repeated themes.

Track the occurrence of each theme and then extract and return the themes that appear more than once.

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.

```python
def identify_repeated_themes(scenes):
  pass
```

Example Usage:

```python
scenes = [
    {"scene": "The hero enters the dark forest.", "theme": "courage"},
    {"scene": "A mysterious figure appears.", "theme": "mystery"},
    {"scene": "The hero faces his fears.", "theme": "courage"},
    {"scene": "An eerie silence fills the air.", "theme": "mystery"},
    {"scene": "The hero finds a hidden treasure.", "theme": "discovery"}
]

repeated_themes = identify_repeated_themes(scenes)
print(repeated_themes)

scenes = [
    {"scene": "The spaceship lands on an alien planet.", "theme": "exploration"},
    {"scene": "A strange creature approaches.", "theme": "danger"},
    {"scene": "The crew explores the new world.", "theme": "exploration"},
    {"scene": "The crew encounters hostile forces.", "theme": "conflict"},
    {"scene": "The crew makes a narrow escape.", "theme": "danger"}
]

repeated_themes = identify_repeated_themes(scenes)
print(repeated_themes)
```

Example Output:

```
['courage', 'mystery']
['exploration', 'danger']
```

Problem 8: Analyze Storyline Continuity

Maintaining a coherent and continuous storyline is crucial for immersion. A storyline may consist of several scenes, each associated with a timestamp that indicates when the event occurs in the narrative. Write a function that, given a list of scene records with timestamps, determines if there are any gaps in the storyline continuity by checking if each scene follows in chronological order.

Iterate through the scenes and verify that the timestamps of consecutive scenes are in increasing order. If any scene is found to be out of sequence, your function should return `False`, indicating a gap in continuity; otherwise, it should return `True`.

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.

```python
def analyze_storyline_continuity(scenes):
  pass
```

Example Usage:

```python
scenes = [
    {"scene": "The hero enters the dark forest.", "timestamp": 1},
    {"scene": "A mysterious figure appears.", "timestamp": 2},
    {"scene": "The hero faces his fears.", "timestamp": 3},
    {"scene": "The hero finds a hidden treasure.", "timestamp": 4},
    {"scene": "An eerie silence fills the air.", "timestamp": 5}
]

continuity = analyze_storyline_continuity(scenes)
print(continuity)

scenes = [
    {"scene": "The spaceship lands on an alien planet.", "timestamp": 3},
    {"scene": "A strange creature approaches.", "timestamp": 2},
    {"scene": "The crew explores the new world.", "timestamp": 4},
    {"scene": "The crew encounters hostile forces.", "timestamp": 5},
    {"scene": "The crew makes a narrow escape.", "timestamp": 6}
]

continuity = analyze_storyline_continuity(scenes)
print(continuity)
```

Example Output:

```
True
False
```

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Track Daily Food Waste

You are given a dictionary where the keys are dates in the format `"YYYY-MM-DD"` and the values are lists of integers representing the amounts of food waste (in grams) recorded on that date. Your task is to calculate the total amount of food waste for each day and return the dates and the total waste amounts for those dates.

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.

```python
def track_daily_food_waste(waste_records):
  pass
```

Example Usage:

```
waste_records1 = {
    "2024-08-01": [200, 150, 50],
    "2024-08-02": [300, 400],
    "2024-08-03": [100]
}

result = track_daily_food_waste(waste_records1)
print(result)

waste_records2 = {
    "2024-07-01": [120, 80],
    "2024-07-02": [150, 200, 50],
    "2024-07-03": [300, 100]
}

result = track_daily_food_waste(waste_records2)
print(result)
```

Example Output:

```
{'2024-08-01': 400, '2024-08-02': 700, '2024-08-03': 100}
{'2024-07-01': 200, '2024-07-02': 400, '2024-07-03': 400}
```

Problem 2: Find Most Wasted Food Item

You are given a dictionary where the keys are food items and the values are lists of integers representing the amounts of each food item wasted (in grams). Your task is to identify which food item was wasted the most frequently in total.

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_most_wasted_food_item(waste_records):
  pass
```

Example Usage:

```
waste_records1 = {
    "Apples": [200, 150, 50],
    "Bananas": [100, 200, 50],
    "Carrots": [150, 100, 200],
    "Tomatoes": [50, 50, 50]
}

result = find_most_wasted_food_item(waste_records1)
print(result)

waste_records2 = {
    "Bread": [300, 400],
    "Milk": [200, 150],
    "Cheese": [100, 200, 100],
    "Fruits": [400, 100]
}

result = find_most_wasted_food_item(waste_records2)
print(result)
```

Example Output:

```
Carrots
Bread
```

Problem 3: Sort Waste Records by Date

You are given a list of tuples where each tuple contains a date (as a string in the format `"YYYY-MM-DD"`) and a list of integers representing the amount of food wasted on that date. Your task is to sort this list by date in ascending order and return the sorted list.

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 sort_waste_records_by_date(waste_records):
  pass
```

Example Usage:

```
waste_records1 = [
    ("2024-08-15", [300, 200]),
    ("2024-08-13", [150, 100]),
    ("2024-08-14", [200, 250]),
    ("2024-08-12", [100, 50])
]

result = sort_waste_records_by_date(waste_records1)
print(result)

waste_records2 = [
    ("2024-07-05", [400, 150]),
    ("2024-07-01", [200, 300]),
    ("2024-07-03", [100, 100]),
    ("2024-07-04", [50, 50])
]

result = sort_waste_records_by_date(waste_records2)
print(result)
```

Example Output:

```
[('2024-08-12', [100, 50]), ('2024-08-13', [150, 100]), ('2024-08-14', [200, 250]),
[('2024-07-01', [200, 300]), ('2024-07-03', [100, 100]), ('2024-07-04', [50, 50]), (
```

Problem 4: Calculate Weekly Waste Totals

You have a dictionary where each key represents a day of the week, and the value for each key is a list of integers representing the amount of food waste (in kilograms) recorded for that day. Your task is to calculate the total food waste for each week and return the results as a dictionary where the keys are the days of the week and the values are the total food waste for each day.

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_weekly_waste_totals(weekly_waste):
  pass
```

Example Usage:

```
weekly_waste = {
    'Monday': [5, 3, 7],
    'Tuesday': [2, 4, 6],
    'Wednesday': [8, 1],
    'Thursday': [4, 5],
    'Friday': [3, 2, 1],
    'Saturday': [6],
    'Sunday': [1, 2, 2]
}
print(calculate_weekly_waste_totals(weekly_waste))
```

Example Output:

```
{'Monday': 15, 'Tuesday': 12, 'Wednesday': 9, 'Thursday': 9, 'Friday': 6, 'Saturday'
```

Problem 5: Filter Records by Waste Threshold

You are given a list of food waste records, where each record is a tuple consisting of a date (in the format `"YYYY-MM-DD"`) and an integer representing the amount of food wasted on that date. You are also given a waste threshold. Your task is to filter out and return a list of tuples with only the records where the waste amount is greater than or equal to the threshold.

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.

```python
def filter_records_by_waste_threshold(waste_records, threshold):
  pass
```

Example Usage:

```python
waste_records1 = [
    ("2024-08-01", 150),
    ("2024-08-02", 200),
    ("2024-08-03", 50),
    ("2024-08-04", 300),
    ("2024-08-05", 100),
    ("2024-08-06", 250)
]
threshold1 = 150

result = filter_records_by_waste_threshold(waste_records1, threshold1)
print(result)

waste_records2 = [
    ("2024-07-01", 90),
    ("2024-07-02", 120),
    ("2024-07-03", 80),
    ("2024-07-04", 130),
    ("2024-07-05", 70)
]
threshold2 = 100

result = filter_records_by_waste_threshold(waste_records2, threshold2)
print(result)
```

Example Output:

```
[('2024-08-01', 150), ('2024-08-02', 200), ('2024-08-04', 300), ('2024-08-06', 250)]
[('2024-07-02', 120), ('2024-07-04', 130)]
```

Problem 6: Track Waste Reduction Trends

You are given a sorted list of daily food waste records where each record is a tuple containing a date (in the format `"YYYY-MM-DD"`) and an integer representing the amount of food wasted on that date. Your task is to determine if there is a trend of reducing food waste over time. Return `True` if each subsequent day shows a decrease in the amount of food wasted compared to the previous day, 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.

```python
def track_waste_reduction_trends(waste_records):
  pass
```

Example Usage:

```python
waste_records_1 = [
    ("2024-08-01", 150),
    ("2024-08-02", 120),
    ("2024-08-03", 100),
    ("2024-08-04", 80),
    ("2024-08-05", 60)
]

waste_records_2 = [
    ("2024-08-01", 150),
    ("2024-08-02", 180),
    ("2024-08-03", 150),
    ("2024-08-04", 140),
    ("2024-08-05", 120)
]

print(track_waste_reduction_trends(waste_records_1))
print(track_waste_reduction_trends(waste_records_2))
```

Example Output:

```
True
False
```

Problem 7: Manage Food Waste

You are tasked with managing food waste records using a queue to simulate the process of handling waste reduction over time. Each record contains a date (in the format `"YYYY-MM-DD"`) and the amount of food wasted on that date. You will process these records using a queue to manage the waste reduction. Return `True` if the total waste in the queue decreases over time as records are processed 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 manage_food_waste_with_queue(waste_records):
  pass
```

Example Usage:

```
waste_records_1 = [
    ("2024-08-01", 150),
    ("2024-08-02", 120),
    ("2024-08-03", 100),
    ("2024-08-04", 80),
    ("2024-08-05", 60)
]

waste_records_2 = [
    ("2024-08-01", 150),
    ("2024-08-02", 180),
    ("2024-08-03", 160),
    ("2024-08-04", 140),
    ("2024-08-05", 120)
]

print(manage_food_waste_with_queue(waste_records_1))
print(manage_food_waste_with_queue(waste_records_2))
```

Example Output:

```
True
False
```

Problem 8: Manage Expiration Dates

Simulate managing food items in a pantry by using a stack to keep track of their expiration dates. Determine if the items are ordered correctly by expiration date (oldest expiration date at the top of the stack). Return `True` if items are ordered correctly 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 check_expiration_order(expiration_dates):
  pass
```

Example Usage:

```
expiration_dates_1 = [
    ("Milk", "2024-08-05"),
    ("Bread", "2024-08-10"),
    ("Eggs", "2024-08-12"),
    ("Cheese", "2024-08-15")
]

expiration_dates_2 = [
    ("Milk", "2024-08-05"),
    ("Bread", "2024-08-12"),
    ("Eggs", "2024-08-10"),
    ("Cheese", "2024-08-15")
]

print(check_expiration_order(expiration_dates_1))
print(check_expiration_order(expiration_dates_2))
```

Example Output:

```
True
False
```

Close Section