


TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: 126663

 **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

While **each problem may call for slightly different approaches to these three steps**, the basics of the steps won't change, and it's important to engage them each time. We've built out some starting points to use in our breakout sessions, below!

Please read the following carefully (take 10 minutes as a team, if you like) and follow these basic steps, as a group, through each of the problems in your problem set.

Fun Fact: We sometimes call the main beats of problem solving, or the tasks that teams are being asked to take, our "DoNow's". If you hear a staff member using this phrase, (e.g., "Ok great! Your team is struggling with the Understand step – what might be a DoNow?") you now know what they might mean!

▼ UPI Example

Step	What is it?	Try It!
1. Understand	Here we strive to Understand what the interviewer is asking for. It's common to restate the problem aloud, ask questions, and consider related test cases.	<ul style="list-style-type: none"> • Nominate one person to share their screen so everyone is on the same page, and bring up the problem at hand. (NOTE: Please trade-off and change who is screen sharing, roughly each problem) • Have one person read the problem aloud. • Have a different person restate the problem in their own words. • Have members of the group ask 2-3 questions about the problem, perhaps about the example usage, or expected output • Example: "Will the list always contain only numbers?"
2. Plan	Then we Plan a solution, starting with appropriate visualizations and pseudocode.	<ul style="list-style-type: none"> • Restate - have one person share the general idea about what the function is trying to accomplish. • Next, break down the problem into subproblems as a group. Each member should participate. <ul style="list-style-type: none"> ◦ If you don't know where to start, try to describe how you would solve the problem <i>without a computer</i>. • As a group, translate each subproblem into pseudocode. <ul style="list-style-type: none"> ◦ How do I do what I described in English in Python? ◦ Then, do I need to change my approach to any steps to make it work in code?
3. Implement	Now we Implement our solution, by translating our Plan into Python.	<ul style="list-style-type: none"> • Translate the pseudocode into Python—this is a stage at which you can consider working individually.

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}, 'Food')
{'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': 2, 'was': 1, 'not': 1, 'amused': 1}  
{'digital': 1, 'nomads': 1, 'love': 1, 'to': 1, 'travel': 2, 'is': 1, 'their': 1, 'passion': 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>`, ``, 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", "WeWork"]
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"), ("New York", "2024-09-01")]
print(most_popular_destination(visits))

visits_2 = [("London", "2024-06-01"), ("Berlin", "2024-06-15"), ("London", "2024-07-01"), ("Paris", "2024-07-15"), ("Tokyo", "2024-08-01")]
print(most_popular_destination(visits_2))

visits_3 = [("Sydney", "2024-05-01"), ("Dubai", "2024-05-15"), ("Sydney", "2024-05-20"), ("Dubai", "2024-06-01"), ("Singapore", "2024-06-15")]
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", "Health", 45),  
print(unique_genres_with_min_length([("Episode A", "Science", 40), ("Episode B", "Science",  
print(unique_genres_with_min_length([("Episode X", "Music", 20), ("Episode Y", "Music", 15),
```



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 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

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.

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

Example Usage:


```

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', 'Climax'),
[('2023-07-01', 'Setup'), ('2023-07-05', 'Opening'), ('2023-07-07', 'Conflict'), ('2023-07-10', 'Climax')]

```

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.

```

def filter_scenes_by_keyword(scenes, keyword):
    pass

```

Example Usage:

```

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 decision.', 'Character grows stronger.', 'Character achieves goal.', '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.']

```



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.

```

def identify_repeated_themes(scenes):
    pass

```

Example Usage:

```

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.

```

def analyze_storyline_continuity(scenes):
    pass

```

Example Usage:

```

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.

```

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]), ('2024-08-15', [300, 200]),
 ('2024-07-01', [200, 300]), ('2024-07-03', [100, 100]), ('2024-07-04', [50, 50]), ('2024-07-05', [400, 150])]

```

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': 6, 'Sunday': 6}
```

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.

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

Example Usage:

```

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.

```

def track_waste_reduction_trends(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", 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