TIP102 | Intermediate Technical Interview Prep

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

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

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.	• 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.	• 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.
		  ○ If you don't know where to start, try to describe how you would solve the problem *without a computer*.
		• As a group, translate each subproblem into pseudocode.
		  ○ 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.	• Translate the pseudocode into Python—this is a stage at which you can consider working individually.

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:

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

Example Output:

```
True
True
False
```

::ai

▼ ✨ AI Hint: Stacks

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **stack**, a data structure that follows the Last In, First Out (LIFO) principle.

If you are unfamiliar with stacks, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a stack is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a stack different from a list or a queue? Can you show me examples of each?"

:::

▼ 💡 **Hint: Pseudocode**

1. Initialize a stack to keep track of the opening tags as you encounter them.
2. Iterate through the posts
    1. If it's an opening tag, push it onto the stack
    2. If it's a closing tag, check if the stack is not empty and whether the tag at the top of the stack is the corresponding opening tag
        1. If yes, pop the opening tag from the stack (we've found its match!)
        2. If no, the tags are not properly nested and we should return `False`
    3. After processing all characters, if the stack is empty, all tags were properly nested and we should return `True`. If the stack is not empty, some opening tags were not

closed, so return `False`

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.

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

Example Usage:

```
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.

```
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**

> This problem may benefit from a algorithmic technique called the **two pointer approach**. Take a look at the unit cheatsheet for a more in-depth overview of how this technique works.

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.

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

Example Usage:

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

Example Output:

```
post

s
```

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

How do we know which data structure and/or algorithm to use when solving this problem? Should we use a stack? A queue? Two pointer? None of the above?

For this problem, a stack would be a good choice because we're checking for the 'balance' of pairs of symbols:

- **Checking and Removing Pairs**
  - As we traverse the string, we can look at each letter one by one.
  - If the letter we're looking at can pair up with the last letter you added to the stack (like `"aA"`), we can remove that last letter from the stack. This is like taking the top plate off the pile.
  - If it doesn't form a pair, we can add the new letter on top of the stack, like putting another plate on top.
- **Rechecking After Removal**

- After we remove a pair, the stack might have a new top letter that could potentially form another pair with the next letter we examine. A stack allows us to handle this smoothly because we only ever look at the top of the stack and the next letter.

For more information about common use cases of stacks, queues, and two pointer, take a look at the unit cheatsheet.

▼ 💡 **Hint: Useful Built-In Methods**

How do we check if we have a lower and uppercase pair? There are many possible ways to accomplish this. Flex your research skills and look up built-in Python functions and string methods to discover different options.

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.

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

Example Usage:

```
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
```

::ai

▼ ✨ **AI Hint: Queues**

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **queue**, a data structure that follows the First In, First Out (FIFO) principle.

If you are unfamiliar with queues, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a queue is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a queue different from a list or stack? Can you show me examples of each?"

:::

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.

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

Example Usage:

```
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**

This problem may benefit from a helper function! If you find your functions getting too long or performing lots of different tasks, it might be a good indicator that you should add a helper function. Helper functions are functions we write to implement a subtask of our primary task. To learn more about helper functions (and inner functions, which is a common way to implement helper functions in Python), check out the unit cheatsheet!

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
```

::ai

▼ ✨ AI Hint: Queues

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **queue**, a data structure that follows the First In, First Out (FIFO) principle.

If you are unfamiliar with queues, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a queue is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a queue different from a list or stack? Can you show me examples of each?"

:::

▼ 💡 **Hint: Pseudocode**

1. Initialize an empty queue.
2. Iterate over the movies list and append a tuple `(i, movies[i])` to the queue where `i` is the user's index, and `movies[i]` is the number of movies they want to stream.
3. While there are still users in the queue:
    1. Process each user by removing them from the front of the queue and icnremnting time by 1 for each movie streamed.
    2. If the user is `k` and has just streamed their last movie, return the current `time`.
    3. If the user still has movies left to stream, put them back at the end of the queue with one less movie.

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**

This problem may benefit from a algorithmic technique called the **two pointer approach**. Take a look at the unit cheatsheet for a more in-depth overview of how this technique works.

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
```

::ai

▼ ✨ AI Hint: Stacks

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **stack**, a data structure that follows the Last In, First Out (LIFO) principle.

If you are unfamiliar with stacks, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a stack is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a stack different from a list or a queue? Can you show me examples of each?"

:::

▼ 💡 **Hint: Pseudocode**

1. Initialize an empty stack to keep track of the characters.

2. Iterate through each character in the `schedule` string:
    1. If the stack is not empty and the top character of the stack is the same as the current character, pop the stack (remove the duplicate pair).
    2. If the top character is not the same as the current character, push the current character onto the stack.
3. After processing all characters, the stack will contain the final schedule without adjacent duplicates.
4. Convert the stack back to a string and return it as the final schedule.

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 that is ear
   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.
```

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

Example Usage:

```
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

Problem 1: Arrange Guest Arrival Order

You are organizing a prestigious event, and you must arrange the order in which guests arrive based on a set of instructions.

The instructions are provided as a 0-indexed string `arrival_pattern` of length `n`, consisting of the characters:

- `'I'` - The next guest should have a **higher** number than the previous guest.
- `'D'` - The next guest should have a **lower** number than the previous guest.

You need to create a string `guest_order` of length `n + 1` that satisfies the following conditions:

1. `guest_order` contains each number from `1` to `str(n + 1)` exactly once. These numbers represent the guests' assigned numbers.
2. For every index `i` from `0` to `n - 1`:
   - If `arrival_pattern[i] == 'I'`, then `guest_order[i] < guest_order[i + 1]`.
   - If `arrival_pattern[i] == 'D'`, then `guest_order[i] > guest_order[i + 1]`.
3. Among all valid orders, return the lexicographically smallest one.

```
def arrange_guest_arrival_order(arrival_pattern):
  pass
```

Example Usage:

```
print(arrange_guest_arrival_order("IIIDIDDD"))
print(arrange_guest_arrival_order("DDD"))
```

Example Output:

```
123549876
4321
```

::ai

▼ ✨ AI Hint: Stacks

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **stack**, a data structure that follows the Last In, First Out (LIFO) principle.

If you are unfamiliar with stacks, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a stack is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a stack different from a list or a queue? Can you show me examples of each?"

:::

Problem 2: Reveal Attendee List in Order

You are organizing an event where each attendee has a unique registration number. These numbers are provided in the list `attendees`.

The attendees will be revealed one by one following a specific process:

1. Reveal the number at the front of the list and remove it.
2. If there are still numbers remaining, take the next number from the front and move it to the back of the list.
3. Repeat this process until all numbers have been revealed.

Your task is to return an ordering of the registration numbers that, when this process is followed, results in the numbers being revealed in **increasing order**.

```
def reveal_attendee_list_in_order(attendees):
  pass
```

Example Usage:

```
print(reveal_attendee_list_in_order([17,13,11,2,3,5,7]))
print(reveal_attendee_list_in_order([1,1000]))
```

Example Output:

```
[2,13,3,11,5,17,7]
[1,1000]
```

::ai

▼ ✨ AI Hint: Queues

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **queue**, a data structure that follows the First In, First Out (FIFO) principle.

If you are unfamiliar with queues, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a queue is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a queue different from a list or stack? Can you show me examples of each?"

:::

Problem 3: Arrange Event Attendees by Priority

You are organizing a large event and need to arrange the attendees based on their priority levels. You are given a 0-indexed list `attendees`, where each element represents the priority level of an attendee, and an integer `priority` that indicates a particular level of priority.

Your task is to rearrange the `attendees` list such that the following conditions are met:

1. Every attendee with a priority less than the specified `priority` appears before every attendee with a priority greater than the specified `priority`.
2. Every attendee with a `priority` equal to the specified priority appears between the attendees with lower and higher priorities.
3. The relative order of the attendees within each priority group (less than, equal to, greater than) must be preserved.

Return the `attendees` list after the rearrangement.

```
def arrange_attendees_by_priority(attendees, priority):
  pass
```

Example Usage:

```
print(arrange_attendees_by_priority([9,12,5,10,14,3,10], 10))
print(arrange_attendees_by_priority([-3,4,3,2], 2))
```

Example Output:

```
[9,5,3,10,10,12,14]
[-3,2,4,3]
```

▼ 💡 **Hint: Two Pointer Variation**

Knowing how to use the basic version of techniques like two pointer is great, but as we begin solving harder problems, we often need to modify them to work in new situations.

For this problem, try using a variation of the two pointer technique where you maintain two pointers as well as a third pointer/iterator `i` to maintain three sections of the array:

- less than priority
- equal to priority
- greater than priority

Problem 4: Rearrange Guests by Attendance and Absence

You are organizing an event, and you have a 0-indexed list `guests` of even length, where each element represents either an attendee (positive integers) or an absence (negative integers). The list contains an equal number of attendees and absences.

You should return the `guests` list rearranged to satisfy the following conditions:

1. Every consecutive pair of elements must have opposite signs, indicating that each attendee is followed by an absence or vice versa.
2. For all elements with the same sign, the order in which they appear in the original list must be preserved.
3. The rearranged list must begin with an attendee (positive integer).

Return the rearranged list after organizing the guests according to the conditions.

```
def rearrange_guests(guests):
  pass
```

Example Usage:

```
print(rearrange_guests([3,1,-2,-5,2,-4]))
print(rearrange_guests([-1,1]))
```

Example Output:

```
[3,-2,1,-5,2,-4]
[1,-1]
```

Problem 5: Minimum Changes to Make Schedule Balanced

You are organizing a series of events, and each event is represented by a parenthesis in the string `schedule`, where an opening parenthesis `(` represents the start of an event, and a closing parenthesis `)` represents the end of an event. A balanced schedule means every event that starts has a corresponding end.

However, due to some scheduling issues, the current `schedule` might not be balanced. In one move, you can insert either a start or an end at any position in the `schedule`.

Return the minimum number of moves required to make the `schedule` balanced.

```
def min_changes_to_make_balanced(schedule):
  pass
```

Example Usage:

```
print(min_changes_to_make_balanced("())"))
print(min_changes_to_make_balanced("((("))
```

Example Output:

```
1
3
```

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

How do we know which data structure and/or algorithm to use when solving this problem? Should we use a stack? A queue? Two pointer? None of the above?

For this problem, a stack would be a good choice because we're checking for the 'balance' of pairs of symbols.

For more information about common use cases of stacks, queues, and two pointers, take a look at the unit cheatsheet.

Problem 6: Marking the Event Timeline

You are organizing a large event, and you need to mark the timeline for a series of scheduled activities.

You are given two strings:

- `event`: A short string representing an event name.
- `timeline`: A longer string representing the full timeline for the event.

Initially, the timeline is empty and represented by a string `t` of the same length as `timeline`, where every character is `'?'`.

In one turn, you can "mark" the timeline by placing the `event` string over any valid position in `t` and copying its letters onto `t`. This replaces the corresponding `'?'` characters in `t`.

Rules:

- You can only place `event` where it fully fits within `t`.
- Each time you mark the timeline, the corresponding letters in `t` are updated.
- Your goal is to perform a sequence of marks so that `t` becomes exactly equal to `timeline`.
- You may use at most `10 * len(timeline)` marks.

Return a list of the starting indices where you placed the `event` string during each mark. If it is impossible to turn `t` into `timeline` following these rules, return an empty list.

```
def mark_event_timeline(event, timeline):
  pass
```

Example Usage:

```
print(mark_event_timeline("abc", "ababc"))
print(mark_event_timeline("abca", "aabcaca"))
```

Example Output:

```
[0, 2]
[3, 0, 1]
```

Explanation

For `"ababc"`:

- Start with `t = "?????"`
- Place `"abc"` at index `0` → `t = "abc??"`
- Place `"abc"` at index `2` → `t = "ababc"` — timeline is complete.

▼ 💡 **Hint: Pseudocode**

1. Start with the initial state of `t` (a string of `?` characters) and add it to the queue
    1. Each element in the queue should be a tuple representing the current state of `t` and the list of indices where `event` has been placed
2. Process the queue
    1. Dequeue an element to get the current state of `t`
    2. For each possible position, try to place `event` at that position

3. If placing `event` helps in moving closer to `timeline`, enqueue the new state of `t` along with the updated list of indices
4. If the current state matches `timeline`, return the list of indices

3. If the queue is exhausted or the number of turns exceeds the limit, return an empty list

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Extra Treats

At the pet adoption center, there are two groups of volunteers:

- `'C'` — Cat Lovers
- `'D'` — Dog Lovers

Each week, these groups vote to decide which type of pet should receive extra treats. The voting happens in rounds, following these rules:

1. **Ban a Vote:** In each round, a volunteer can ban one volunteer from the opposite group. A banned volunteer loses all voting rights for the rest of the process.
2. **Declare Victory:** If at any point all remaining volunteers are from the same group, that group can declare victory, and their preferred pet will receive the extra treats.

You are given a string `votes` representing the group affiliation of each volunteer. The character at index `i` is either `'C'` (Cat Lovers) or `'D'` (Dog Lovers).

Assuming each volunteer acts in order (from left to right) and the process repeats until one group wins, predict which group will eventually declare victory.

Return:

- `"Cat Lovers"` if the Cat Lovers will win.
- `"Dog Lovers"` if the Dog Lovers will win.

```
def predictAdoption_victory(votes):
  pass
```

Example Usage:

```
print(predictAdoption_victory("CD"))
print(predictAdoption_victory("CDD"))
```

Example Output:

```
Cat Lovers
Dog Lovers
```

::ai

▼ ✨ AI Hint: Queues

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **queue**, a data structure that follows the First In, First Out (FIFO) principle.

If you are unfamiliar with queues, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a queue is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a queue different from a list or stack? Can you show me examples of each?"

:::

Problem 2: Pair Up Animals for Shelter

In an animal shelter, animals are paired up to share a room. Each pair has a discomfort level, which is the sum of their individual discomfort levels. The shelter's goal is to minimize the maximum discomfort level among all pairs to ensure that the rooms are as comfortable as possible.

Given a list `discomfort_levels` representing the discomfort levels of `n` animals, where `n` is even, pair up the animals into `n / 2` pairs such that:

1. Each animal is in exactly one pair, and
2. The maximum discomfort level among the pairs is minimized. Return the minimized maximum discomfort level after optimally pairing up the animals.

Return the minimized maximum comfort level after optimally pairing up the animals.

```
def pair_up_animals(discomfort_levels):
  pass
```

Example Usage:

```
print(pair_up_animals([3,5,2,3]))
print(pair_up_animals([3,5,4,2,4,6]))
```

Example Output:

```
7
8
```

▼ 💡 **Hint: Two Pointer Technique**

This problem may benefit from a algorithmic technique called the **two pointer approach**. Take a look at the unit cheatsheet for a more in-depth overview of how this technique works.

Problem 3: Rearrange Animals and Slogans

You are given a string `s` that consists of lowercase English letters representing animal names or slogans and brackets. The goal is to rearrange the animal names or slogans in each pair of matching parentheses by reversing them, starting from the innermost pair.

After processing, your result should not contain any brackets.

```
def rearrange_animal_names(s):
  pass
```

Example Usage:

```
print(rearrange_animal_names("(dribtacgod)"))
print(rearrange_animal_names("(!(love(stac))I)"))
print(rearrange_animal_names("adopt(yadot(a(tep)))!"))
```

Example Output:

```
dogcatbird
Ilovecats!
adoptapettoday!
```

::ai

▼ ✨ **AI Hint: Stacks**

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **stack**, a data structure that follows the Last In, First Out (LIFO) principle.

If you are unfamiliar with stacks, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a stack is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a stack different from a list or a queue? Can you show me examples of each?"

:::

Problem 4: Append Animals to Include Preference

You are managing an animal adoption center. You have:

- A string `available`, representing the animals currently available for adoption.
- A string `preferred`, representing a customer's preferred sequence of animals.

You want to make sure the `preferred` sequence appears as a **subsequence** of `available`. A subsequence means the characters appear in order, but not necessarily consecutively.

To achieve this, you are allowed to append characters to the **end** of `available`. You cannot remove characters or insert them elsewhere.

Return the **minimum number of characters** you need to append to the end of `available` so that `preferred` becomes a subsequence.

```
def append_animals(available, preferred):
  pass
```

Example Usage:

```
print(append_animals("catsdogs", "cows"))
print(append_animals("rabbit", "r"))
print(append_animals("fish", "bird"))
```

Example Output:

```
2
0
4
```

Explanation:

`available = "catsdogs"` `preferred = "cows"`

- `'c'` → found at index 0
- `'o'` → found at index 5
- `'w'` → not present

You need to append `"ws"` to the end to complete the subsequence. **Minimum characters to append:** `2`

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

How do we know which data structure and/or algorithm to use when solving this problem? Should we use a stack? A queue? Two pointer? None of the above?

For this problem, two pointer would be a good choice because we're comparing and matching elements from two sequences in a linear fashion.

For more information about common use cases of stacks, queues, and two pointer, take a look at the unit cheatsheet.

Problem 5: Group Animals by Habitat

You are managing a wildlife sanctuary where animals of the same species need to be grouped together by their habitats. Given a string `habitats` representing the sequence of animals, where each character corresponds to a particular species, you need to partition the string into as many contiguous groups as possible, ensuring that each species appears in at most one group.

The order of species in the resultant sequence must remain the same as in the input string `habitats`.

Return a list of integers representing the size of these habitat groups.

```
def group_animals_by_habitat(habitats):
  pass
```

Example Usage:

```
print(group_animals_by_habitat("ababcbacadefegdehijhklij"))
print(group_animals_by_habitat("eccbbbbdec"))
```

Example Output:

```
[9,7,8]
[10]
```

Problem 6: Validate Animal Sheltering Sequence

Given two integer arrays `admitted` and `adopted` each with distinct values representing animals in an animal shelter, return `True` if this could have been the result of a sequence of admitting and adopting animals from the shelter, or `False` otherwise.

```
def validate_shelter_sequence(admitted, adopted):
  pass
```

Example Usage:

```
print(validate_shelter_sequence([1,2,3,4,5], [4,5,3,2,1]))
print(validate_shelter_sequence([1,2,3,4,5], [4,3,5,1,2]))
```

Example Output:

```
True
False
```

Close Section