TIP102 | Intermediate Technical Interview Prep

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

Session 1: Stacks, Queues, and Two Pointer

Session Overview

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

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

🎢 Part 1: Instructor Led Session

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

👨‍💻 Part 2: Breakout Session

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

In breakout rooms you will:

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

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

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

▶ **Note on Expectations**

🔎 Problem Solving Approach

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

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

We will learn to:

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

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 1

▶ Standard Problem Set Version 1

▶ Standard Problem Set Version 2

▼ 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 their status. The sequence is dictated by a 0-indexed string `arrival_pattern` of length `n`, consisting of the characters `'I'` meaning the next guest should have a higher status than the previous one, and `'D'` meaning the next guest should have a lower status than the previous one.

You need to create a 0-indexed string `guest_order` of length `n + 1` that satisfies the following conditions:

- `guest_order` consists of the digits `'1'` to `'9'`, where each digit represents the guest's status and is used at most once.
- 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]`.

Return the lexicographically smallest possible string guest_order that meets the conditions.

```
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
```

▶ 💡 **Hint: Stacks**

Problem 2: Reveal Attendee List in Order

You are organizing an event where attendees have unique registration numbers. These numbers are provided in the list `attendees`. You need to arrange the attendees in a way that, when their registration numbers are revealed one by one, the numbers appear in increasing order.

The process of revealing the attendee list follows these steps repeatedly until all registration numbers are revealed:

1. Take the top registration number from the list, reveal it, and remove it from the list.
2. If there are still registration numbers in the list, take the next top registration number and move it to the bottom of the list.
3. If there are still unrevealed registration numbers, go back to step 1. Otherwise, stop.

Return an ordering of the registration numbers that would reveal the attendees 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]
```

▶ 💡 **Hint: Queues**

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**

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**

Problem 6: Marking the Event Timeline

You are given two strings `event` and `timeline`. Initially, there is a string `t` of length `timeline.length` with all `t[i] == '?'`.

In one turn, you can place `event` over `t` and replace every letter in `t` with the corresponding letter from `event`.

For example, if `event = "abc"` and `timeline = "abcba"`, then `t` is `"?????"` initially. In one turn, you can:

- place `event` at index 0 of `t` to obtain `"abc??"`,
- place `event` at index 1 of `t` to obtain `"?abc?"`, or
- place `event` at index 2 of `t` to obtain `"??abc"`.

Note that `event` must be fully contained within the boundaries of `t` in order to mark (i.e., you cannot place `event` at index 3 of `t`). We want to convert `t` to `timeline` using at most `10 * timeline.length` turns.

Return an array of the index of the left-most letter being marked at each turn. If we cannot obtain `timeline` from `t` within `10 * timeline.length` turns, return an empty array.

```
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]
```

▶ 💡 **Hint: Pseudocode**

Close Section

▼ Advanced Problem Set Version 2

Problem 1: Extra Treats

In a pet adoption center, there are two groups of volunteers: the "Cat Lovers" and the "Dog Lovers."

The center is deciding on which type of pet should be receive extra treats that week, and the voting takes place in a round-based procedure. In each round, each volunteer can exercise one of the two rights:

- **Ban one volunteer's vote:** A volunteer can make another volunteer from the opposite group lose all their rights in this and all the following rounds.
- **Announce the victory:** If a volunteer finds that all the remaining volunteers with the right to vote are from the same group, they can announce the victory for their group and prioritize their preferred pet for extra treats.

Given a string `votes` representing each volunteer's group affiliation. The character 'C' represents "Cat Lovers" and 'D' represents "Dog Lovers". The length of the given string represents the number of volunteers.

Predict which group will finally announce the victory and prioritize their preferred pet for extra treats. The output should be `"Cat Lovers"` or `"Dog Lovers"`.

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

Example Usage:

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

Example Output:

```
Cat Lovers
Dog Lovers
```

▶ 💡 **Hint: Queues**

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**

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!
```

▶ 💡 **Hint: Stacks**

Problem 4: Append Animals to Include Preference

You are managing an animal adoption center, and you have two lists of animals: `available` and `preferred`, both consisting of lowercase English letters representing different types of animals.

Return the minimum number of characters that need to be appended to the end of the `available` list so that `preferred` becomes a subsequence of `available`.

A subsequence is a string that can be derived from another string by deleting some or no characters without changing the order of the remaining characters.

```
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
```

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

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