TIP102 | Intermediate Technical Interview Prep

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

Session 2: 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 strengthen their ability to solve problems using stacks, queues, and the two pointer method.

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

🎢 Part 1 : Instructor Led Session

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

👨‍💻 Part 2: Breakout Session

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

In breakout rooms you will:

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

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

▶ **Note on Expectations**

🔎 Problem Solving Approach

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

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

We will learn to:

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

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 2

▼ Standard Problem Set Version 1

Problem 1: Manage Performance Stage Changes

At a cultural festival, multiple performances are scheduled on a single stage. However, due to last-minute changes, some performances need to be rescheduled or canceled. The festival organizers use a stack to manage these changes efficiently.

You are given a list `changes` of strings where each string represents a change action. The actions can be:

- `"Schedule X"`: Schedule a performance with ID X on the stage.
- `"Cancel"`: Cancel the most recently scheduled performance that hasn't been canceled yet.
- `"Reschedule"`: Reschedule the most recently canceled performance to be the next on stage.

Return a list of performance IDs that remain scheduled on the stage after all changes have been applied.

```
def manage_stage_changes(changes):
    pass
```

Example Usage:

```
print(manage_stage_changes(["Schedule A", "Schedule B", "Cancel", "Schedule C", "Res
print(manage_stage_changes(["Schedule A", "Cancel", "Schedule B", "Cancel", "Resched
print(manage_stage_changes(["Schedule X", "Schedule Y", "Cancel", "Cancel", "Schedul
```

Example Output:

```
["A", "C", "B", "D"]
[]
["Z"]
```

▶ 💡 **Hint: Stacks**

Problem 2: Queue of Performance Requests

You are organizing a festival and want to manage the queue of requests to perform. Each request has a priority. Use a queue to process the performance requests in the order they arrive but ensure that requests with higher priorities are processed before those with lower priorities. Return the order in which performances are processed.

```
def process_performance_requests(requests):
    pass
```

Example Usage:

```
print(process_performance_requests([(3, 'Dance'), (5, 'Music'), (1, 'Drama')]))
print(process_performance_requests([(2, 'Poetry'), (1, 'Magic Show'), (4, 'Concert')
print(process_performance_requests([(1, 'Art Exhibition'), (3, 'Film Screening'), (2
```

Example Output:

```
['Music', 'Dance', 'Drama']
['Concert', 'Stand-up Comedy', 'Poetry', 'Magic Show']
['Keynote Speech', 'Panel Discussion', 'Film Screening', 'Workshop', 'Art Exhibition
```

▶ 💡 **Hint: Queues**

Problem 3: Collecting Points at Festival Booths

At the festival, there are various booths where visitors can collect points. Each booth has a specific number of points available. Use a stack to simulate the process of collecting points and return the total points collected after visiting all booths.

```
def collect_festival_points(points):
    pass
```

Example Usage:

```
print(collect_festival_points([5, 8, 3, 10]))
print(collect_festival_points([2, 7, 4, 6]))
print(collect_festival_points([1, 5, 9, 2, 8]))
```

Example Output:

```
26
19
25
```

Problem 4: Festival Booth Navigation

At the cultural festival, you are managing a treasure hunt where participants need to visit booths in a specific order. The order in which they should visit the booths is defined by a series of clues. However, some clues lead to dead ends, and participants must backtrack to previous booths to continue their journey.

You are given a list of clues, where each clue is either a booth number (an integer) to visit or the word "back" indicating that the participant should backtrack to the previous booth.

Write a function to simulate the participant's journey and return the final sequence of booths visited, in the order they were visited.

```
def booth_navigation(clues):
    pass
```

Example Usage:

```
clues = [1, 2, "back", 3, 4]
print(booth_navigation(clues))

clues = [5, 3, 2, "back", "back", 7]
print(booth_navigation(clues))

clues = [1, "back", 2, "back", "back", 3]
print(booth_navigation(clues))
```

Example Output:

```
[1, 3, 4]
[5, 7]
[3]
```

Problem 5: Merge Performance Schedules

You are organizing a cultural festival and have two performance schedules, schedule1 and schedule2, each represented by a string where each character corresponds to a performance slot. Merge the schedules by adding performances in alternating order, starting with schedule1. If one schedule is longer than the other, append the additional performances onto the end of the merged schedule.

Return the merged performance schedule.

```
def merge_schedules(schedule1, schedule2):
    pass
```

Example Usage:

```
print(merge_schedules("abc", "pqr"))
print(merge_schedules("ab", "pqrs"))
print(merge_schedules("abcd", "pq"))
```

Example Output:

```
apbqcr
apbqrs
apbqcd
```

▶ 💡 **Hint: Two Pointer Technique**

Problem 6: Next Greater Event

At a cultural festival, you have a schedule of events where each event has a unique popularity score. The schedule is represented by two distinct 0-indexed integer arrays `schedule1` and `schedule2`, where `schedule1` is a subset of `schedule2`.

For each event in `schedule1`, find its position in `schedule2` and determine the next event in `schedule2` with a higher popularity score. If there is no such event, then the answer for that event is `-1`.

Return an array `ans` of length `schedule1.length` such that `ans[i]` is the next greater event's popularity score as described above.

```
def next_greater_event(schedule1, schedule2):
  pass
```

Example Usage:

```
print(next_greater_event([4, 1, 2], [1, 3, 4, 2]))
print(next_greater_event([2, 4], [1, 2, 3, 4]))
```

Example Output:

```
[-1, 3, -1]
[3, -1]
```

Problem 7: Sort Performances by Type

You are organizing a cultural festival and have a list of performances represented by an integer array `performances`. Each performance is classified as either an even type (e.g., dance, music) or an odd type (e.g., drama, poetry).

Your task is to rearrange the performances so that all the even-type performances appear at the beginning of the array, followed by all the odd-type performances.

Return any array that satisfies this condition.

```
def sort_performances_by_type(performances):
  pass
```

Example Usage:

```
print(sort_performances_by_type([3, 1, 2, 4]))
print(sort_performances_by_type([0]))
```

Example Output:

```
[4, 2, 1, 3]
[0]
```

Close Section

▾ Standard Problem Set Version 2

Problem 1: Final Costs After a Supply Discount

You are managing the budget for a global expedition, where the cost of supplies is represented by an integer array `costs`, where `costs[i]` is the cost of the `i` th supply item.

There is a special discount available during the expedition. If you purchase the `i` th item, you will receive a discount equivalent to `costs[j]`, where `j` is the minimum index such that `j > i` and `costs[j] <= costs[i]`. If no such `j` exists, you will not receive any discount.

Return an integer array `final_costs` where `final_costs[i]` is the final cost you will pay for the `i` th supply item, considering the special discount.

```
def final_supply_costs(costs):
  pass
```

Example Usage:

```
print(final_supply_costs([8, 4, 6, 2, 3]))
print(final_supply_costs([1, 2, 3, 4, 5]))
print(final_supply_costs([10, 1, 1, 6]))
```

Example Output:

```
[4, 2, 4, 2, 3]
[1, 2, 3, 4, 5]
[9, 0, 1, 6]
```

▶ 💡 **Hint: Stacks**

Problem 2: Find First Symmetrical Landmark Name

During your global expedition, you encounter a series of landmarks, each represented by a string in the array `landmarks`. Your task is to find and return the first symmetrical landmark name. If there is no such name, return an empty string `""`.

A landmark name is considered symmetrical if it reads the same forward and backward.

```
def first_symmetrical_landmark(landmarks):
  pass
```

Example Usage:

```
print(first_symmetrical_landmark(["canyon","forest","rotor","mountain"]))
print(first_symmetrical_landmark(["plateau","valley","cliff"]))
```

Example Output:

```
rotor
```

▶ 💡 **Hint: Two Pointer Technique**

▶ 💡 **Hint: Helper Functions**

Problem 3: Terrain Elevation Match

During your global expedition, you are mapping out the terrain elevations, where the elevation of each point is represented by an integer. You are given a string `terrain` of length `n`, where:

- `terrain[i] == 'I'` indicates that the elevation at the `i`th point is lower than the elevation at the `i`+1th point (`elevation[i] < elevation[i + 1]`).

- `terrain[i] == 'D'` indicates that the elevation at the `i`th point is higher than the elevation at the `i`+1th point (`elevation[i] > elevation[i + 1]`).

Your task is to reconstruct the elevation sequence and return it as a list of integers. If there are multiple valid sequences, return any of them.

Hint: Try using two variables: one to track the smallest available number and one for the largest. As you process each character in the string, assign the smallest number when the next elevation should increase ('I'), and assign the largest number when the next elevation should decrease ('D').

```
def terrain_elevation_match(terrain):
  pass
```

Example Usage:

```
print(terrain_elevation_match("IDID"))
print(terrain_elevation_match("III"))
print(terrain_elevation_match("DDI"))
```

Example Output:

```
[0, 4, 1, 3, 2]
[0, 1, 2, 3]
[3, 2, 0, 1]
```

Problem 4: Find the Expedition Log Concatenation Value

You are recording journal entries during a global expedition, where each entry is represented by a 0-indexed integer array, `logs`. The concatenation of two journal entries means combining their numerals into one.

For example, concatenating the numbers 15 and 49 results in 1549.

Your task is to calculate the total concatenation value of all the journal entries, which starts at 0. To do this, perform the following steps until no entries remain:

1. If there are at least two entries in the `logs`, concatenate the first and last entries, add the result to the current concatenation value, and then remove these two entries.
2. If there is only one entry left, add its value to the concatenation value and remove it from the array.
3.

Return the final concatenation value after all entries have been processed.

```
def find_the_log_conc_val(logs):
  pass
```

Example Usage:

```
print(find_the_log_conc_val([7, 52, 2, 4]))
print(find_the_log_conc_val([5, 14, 13, 8, 12]))
```

Example Output:

```
596
673
```

Problem 5: Number of Explorers Unable to Gather Supplies

During a global expedition, explorers must gather supplies from a limited stockpile, which includes two types of resources: type 0 (e.g., food rations) and type 1 (e.g., medical kits). The explorers are lined up in a queue, each with a specific preference for one of the two types of resources.

The number of supplies in the stockpile is equal to the number of explorers. The supplies are stacked in a pile. At each step:

- If the explorer at the front of the queue prefers the resource on the top of the stack, they will take it and leave the queue.
- Otherwise, they will leave the resource and go to the end of the queue.

This process continues until no explorer in the queue wants to take the top resource, leaving some explorers unable to gather the supplies they need.

You are given two integer arrays explorers and supplies, where `supplies[i]` is the type of the `i` th resource in the stack (`i = 0` is the top of the stack) and `explorers[j]` is the preference of the `j` th explorer in the initial queue (`j = 0` is the front of the queue). Return the number of explorers that are unable to gather their preferred supplies.

```
def count_explorers(explorers, supplies):
  pass
```

Example Usage:

```
print(count_explorers([1, 1, 0, 0], [0, 1, 0, 1]))
print(count_explorers([1, 1, 1, 0, 0, 1], [1, 0, 0, 0, 1, 1]))
```

Example Output:

```
0
3
```

Problem 6: Count Balanced Terrain Subsections

During your global expedition, you are analyzing a binary terrain string, `terrain`, where `0` represents a valley and `1` represents a hill. You need to count the number of non-empty *balanced subsections* in the terrain. A balanced subsection is defined as a contiguous segment of the terrain

where an equal number of valleys (`0` s) and hills (`1` s) appear, and all the `0` s and `1` s are grouped consecutively.

Your task is to return the total number of these balanced subsections. Note that subsections that occur multiple times should be counted each time they appear.

```
def count_balanced_terrain_subsections(terrain):
    pass
```

Example Usage:

```
print(count_balanced_terrain_subsections("00110011"))
print(count_balanced_terrain_subsections("10101"))
```

Example Output:

```
6
4
```

Problem 7: Check if a Signal Occurs as a Prefix in Any Transmission

During your global expedition, you are monitoring various transmissions, each consisting of some signals separated by a single space. You are given a `searchSignal` and need to check if it occurs as a prefix to any signal in a transmission.

Return the index of the signal in the transmission (1-indexed) where `searchSignal` is a prefix of this signal. If `searchSignal` is a prefix of more than one signal, return the index of the first signal (minimum index). If there is no such signal, return `-1` .

A prefix of a string `s` is any leading contiguous substring of `s` .

```
def is_prefix_of_signal(transmission, searchSignal):
  pass
```

Example Usage:

```
print(is_prefix_of_signal("i love eating burger", "burg"))
print(is_prefix_of_signal("this problem is an easy problem", "pro"))
print(is_prefix_of_signal("i am tired", "you"))
```

Example Output:

```
4
2
-1
```

Close Section

- **Advanced Problem Set Version 1**
- **Advanced Problem Set Version 2**