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

▶ Standard Problem Set Version 2

▼ Advanced Problem Set Version 1

Problem 1: Blueprint Approval Process

You are in charge of overseeing the blueprint approval process for various architectural designs. Each blueprint has a specific complexity level, represented by an integer. Due to the complex nature of the designs, the approval process follows a strict order:

1. Blueprints with lower complexity should be reviewed first.
2. If a blueprint with higher complexity is submitted, it must wait until all simpler blueprints have been approved.

Your task is to simulate the blueprint approval process using a queue. You will receive a list of blueprints, each represented by their complexity level in the order they are submitted. Process the blueprints such that the simpler designs (lower numbers) are approved before more complex ones.

Return the order in which the blueprints are approved.

```python
def blueprint_approval(blueprints):
    pass
```

Example Usage:

```python
print(blueprint_approval([3, 5, 2, 1, 4]))
print(blueprint_approval([7, 4, 6, 2, 5]))
```

Example Output:

```
[1, 2, 3, 4, 5]
[2, 4, 5, 6, 7]
```

Problem 2: Build the Tallest Skyscraper

You are given an array `floors` representing the heights of different building floors. Your task is to design a skyscraper using these floors, where each floor must be placed on top of a floor with equal or greater height. However, you can only start a new skyscraper when necessary, meaning when no more floors can be added to the current skyscraper according to the rules.

Return the number of skyscrapers you can build using the given floors.

```python
def build_skyscrapers(floors):
    pass
```

Example Usage:

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

Example Output:

```
4
4
2
```

Problem 3: Dream Corridor Design

You are an architect designing a corridor for a futuristic dream space. The corridor is represented by a list of integer values where each value represents the width of a segment of the corridor. Your goal is to find two segments such that the corridor formed between them (including the two segments) has the maximum possible area. The area is defined as the minimum width of the two segments multiplied by the distance between them.

You need to return the maximum possible area that can be achieved.

```python
def max_corridor_area(segments):
    pass
```

Example Usage:

```python
print(max_corridor_area([1, 8, 6, 2, 5, 4, 8, 3, 7]))
print(max_corridor_area([1, 1]))
```

Example Output:

```
49
1
```

Problem 4: Dream Building Layout

You are an architect tasked with designing a dream building layout. The building layout is represented by a string `s` of even length `n`. The string consists of exactly `n / 2` left walls `'['` and `n / 2` right walls `']'`.

A layout is considered balanced if and only if:

- It is an empty space, or
- It can be divided into two separate balanced layouts, or
- It can be surrounded by left and right walls that balance each other out.

You may swap the positions of any two walls any number of times.

Return the minimum number of swaps needed to make the building layout balanced.

```python
def min_swaps(s):
    pass
```

Example Usage:

```python
print(min_swaps("][]["))
print(min_swaps("]]][[["))
print(min_swaps("[]"))
```

Example Output:

```
1
2
0
```

Problem 5: Designing a Balanced Room

You are designing a room layout represented by a string `s` consisting of walls `'('`, `')'`, and decorations in the form of lowercase English letters.

Your task is to remove the minimum number of walls `'('` or `')'` in any positions so that the resulting room layout is balanced and return any valid layout.

Formally, a room layout is considered balanced if and only if:

- It is an empty room (an empty string), contains only decorations (lowercase letters), or
- It can be represented as AB (A concatenated with B), where A and B are valid layouts, or
- It can be represented as (A), where A is a valid layout.

```
def make_balanced_room(s):
    pass
```

Example Usage:

```
print(make_balanced_room("art(t(d)e)sign)"))
print(make_balanced_room("d)e(s)ign"))
print(make_balanced_room("))(("))
```

Example Output:

```
art(t(d)e)s)ign
de(s)ign
```

Problem 6: Time to Complete Each Dream Design

As an architect, you are working on a series of imaginative designs for various dreamscapes. Each design takes a certain amount of time to complete, depending on the complexity of the elements involved. You want to know how many days it will take for each design to be ready for the next one to begin, assuming each subsequent design is more complex and thus takes more time to finish.

You are given an array `design_times` where each element represents the time in days needed to complete a particular design. For each design, determine the number of days you will have to wait until a more complex design (one that takes more days) is ready to begin. If no such design exists for a particular design, return `0` for that position.

Return an array `answer` such that `answer[i]` is the number of days you have to wait after the `i`-th design to start working on a more complex design. If there is no future design that is more complex, keep `answer[i] == 0` instead.

```
def time_to_complete_dream_designs(design_times):
    pass
```

Example Usage:

```
print(time_to_complete_dream_designs([3, 4, 5, 2, 1, 6, 7, 3]))
print(time_to_complete_dream_designs([2, 3, 1, 4]))
print(time_to_complete_dream_designs([5, 5, 5, 5]))
```

Example Output:

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

Problem 7: Next Greater Element

You are designing a sequence of dream elements, each represented by a number. The sequence is circular, meaning that the last element is followed by the first. Your task is to determine the next greater dream element for each element in the sequence.

The next greater dream element for a dream element `x` is the first element that is greater than x when traversing the sequence in its natural circular order. If no such dream element exists, return -1 for that dream element.

```
def next_greater_dream(dreams):
    pass
```

Example Usage:

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

Example Output:

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

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Score of Mystical Market Chains

In the mystical market, chains of magical items are represented by a string of balanced symbols. The score of these chains is determined by the mystical power within the string, following these rules:

- The symbol `"()"` represents a basic magical item with a power score of 1.
- A chain `AB`, where `A` and `B` are balanced chains of magical items, has a total power score of `A + B`.
- A chain (`A`), where `A` is a balanced chain of magical items, has a power score of `2 * A`.

Given a balanced string representing a chain of magical items, return the total power score of the chain.

```
def score_of_mystical_market_chains(chain):
  pass
```

Example Usage:

```
print(score_of_mystical_market_chains("()"))
print(score_of_mystical_market_chains("(())"))
print(score_of_mystical_market_chains("()()"))
```

Example Output:

```
1
2
2
```

Problem 2: Arrange Magical Orbs

In the mystical market, you have a collection of magical orbs, each of which is colored red, white, or blue. Your task is to arrange these orbs in a specific order so that all orbs of the same color are adjacent to each other. The colors should be ordered as red, white, and blue.

We will use the integers 0, 1, and 2 to represent the colors red, white, and blue, respectively.

You must arrange the orbs in-place without using any library's sorting function.

```
def arrange_magical_orbs(orbs):
  pass
```

Example Usage:

```
orbs1 = [2, 0, 2, 1, 1, 0]
arrange_magical_orbs(orbs1)
print(orbs1)

orbs2 = [2, 0, 1]
arrange_magical_orbs(orbs2)
print(orbs2)
```

Example Output:

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

Problem 3: Matching of Buyers with Sellers

In the mystical market, you are given a list of buyers, where each buyer has a specific amount of gold to spend. You are also given a list of sellers, where each seller has a specific price for their magical goods.

A buyer can purchase from a seller if the buyer's gold is greater than or equal to the seller's price. Additionally, each buyer can make at most one purchase, and each seller can sell their goods to at most one buyer.

Return the maximum number of transactions that can be made in the market that satisfy these conditions.

```
def match_buyers_and_sellers(buyers, sellers):
  pass
```

Example Usage:

```
buyers1 = [4, 7, 9]
sellers1 = [8, 2, 5, 8]
print(match_buyers_and_sellers(buyers1, sellers1))

buyers2 = [1, 1, 1]
sellers2 = [10]
print(match_buyers_and_sellers(buyers2, sellers2))
```

Example Output:

```
3
0
```

Problem 4: Maximum Value from Removing Rare Items

In the Mystical Market, you are given a collection of mystical items in a string format `items` and two integers `x` and `y`. You can perform two types of operations any number of times to remove rare item pairs and gain value.

- Remove the pair of items `"AB"` and gain `x` value points.
- Remove the pair of items `"BA"` and gain `y` value points.

Return the maximum value you can gain after applying the above operations on items.

```
def maximum_value(items, x, y):
  pass
```

Example Usage:

```
s1 = "cdbcbbaaabab"
x1, y1 = 4, 5
print(maximum_value(s1, x1, y1))

s2 = "aabbaaxybbaabb"
x2, y2 = 5, 4
print(maximum_value(s2, x2, y2))
```

Example Output:

```
19
20
```

Problem 5: Strongest Magical Artifacts

In the Mystical Market, you are given an array of magical artifacts represented by integers `artifacts`, and an integer `k`.

A magical artifact `artifacts[i]` is said to be stronger than `artifacts[j]` if `|artifacts[i] - m| > |artifacts[j] - m|` where `m` is the median strength of the artifacts. If `|artifacts[i] - m| == |artifacts[j] - m|`, then `artifacts[i]` is said to be stronger than `artifacts[j]` if `artifacts[i] > artifacts[j]`.

Return a list of the strongest `k` magical artifacts in the Mystical Market. Return the answer in any arbitrary order.

```python
def get_strongest_artifacts(artifacts, k):
  pass
```

Example Usage:

```python
print(get_strongest_artifacts([1, 2, 3, 4, 5], 2))
print(get_strongest_artifacts([1, 1, 3, 5, 5], 2))
print(get_strongest_artifacts([6, 7, 11, 7, 6, 8], 5))
```

Example Output:

```
[5, 1]
[5, 5]
[11, 8, 6, 6, 7]
```

Problem 6: Enchanted Boats

You are given an array creatures where `creatures[i]` is the magical power of the `i` th creature, and an infinite number of enchanted boats where each boat can carry a maximum magical load of `limit`. Each boat carries at most two creatures at the same time, provided the sum of the magical power of those creatures is at most `limit`.

Return the minimum number of enchanted boats required to carry every magical creature.

```python
def num_enchanted_boats(creatures, limit):
  pass
```

Example Usage:

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

Example Output:

```
1
3
4
```

Problem 7: Market Token Value

You are a vendor in a mystical market where magical tokens are used for trading. The value of a token is determined by its structure, represented by a string containing pairs of mystical brackets `()`.

The value of a mystical token is calculated based on the following rules:

- `()` has a value of 1.
- The value of two adjacent tokens `AB` is the sum of their individual values, where `A` and `B` are valid token structures.
- The value of a nested token `(A)` is twice the value of the token inside the brackets.

Your task is to calculate the total value of a given mystical token string.

```
def token_value(token):
  pass
```

Example Usage:

```
print(token_value("()"))
print(token_value("(())"))
print(token_value("()()"))
```

Example Output:

```
1
2
2
```

Close Section