# 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: Binary Search and Divide and Conquer

## Session Overview

This session covers binary search, recursion, and divide-and-conquer techniques to solve problems efficiently. Tasks include finding cruise lengths, locating cabins, counting checked-in passengers, and determining profitability of excursions. Other challenges involve finding the shallowest route point, conducting a treasure hunt in a grid, and solving music-related problems, all emphasizing optimal performance with logarithmic time complexity.

> 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: Find Millenium Falcon Part

Han Solo's ship, the Millenium Falcon, has broken down and he's searching for a specific replacement part. As a repair shop owner helping him out, write a function `check_stock()` that takes in a list `inventory` where each element is an integer ID of a part you stock in your shop, and an integer `part_id` representing the integer ID of the part Han Solo is looking for. Return `True` if the `part_id` is in `inventory` and `False` otherwise.

Your solution must have `O(log n)` time complexity.

```python
def check_stock(inventory, part_id):
    pass
```

Example Usage:

```python
print(check_stock([1, 2, 5, 12, 20], 20))
print(check_stock([1, 2, 5, 12, 20], 100))
```

Example Ouput:

```
True
False
```

▶ 💡 **Hint: Binary Search**

▶ 💡 **Hint: `O(log n)` Time Complexity**

## Problem 2: Find Millenium Falcon Part II

If you implemented your `check_stock()` function from the previous problem iteratively, implement it recursively. If you implemented it recursively, implement it iteratively.

```python
def check_stock(inventory, part_id):
    pass
```

Example Usage:

```python
print(check_stock([1, 2, 5, 20, 12], 20))
print(check_stock([1, 2, 5, 20, 12], 100))
```

Example Ouput:

```
True
False
```

## Problem 3: Find First and Last Frequency Positions

The Rebel Alliance has intercepted a crucial sequence of encrypted transmissions from the evil Empire. Each transmission is marked with a unique frequency code, represented as integers, and these codes are stored in a sorted array `transmissions`. As a skilled codebreaker for the Rebellion, write a function `find_frequency_positions()` that returns a tuple with the first and last indices of a specific frequency code `target_code` in `transmissions`. If `target_code` does not exist in `transmissions`, return `(-1, -1)`.

Your solution must have `O(log n)` time complexity.

```python
def find_frequency_positions(transmissions, target_code):
    pass
```

Example Usage:

```python
print(find_frequency_positions([5,7,7,8,8,10], 8))
print(find_frequency_positions([5,7,7,8,8,10], 6))
print(find_frequency_positions([], 0))
```

Example Output:

```
(3, 4)
(-1, -1)
(-1, -1)
```

## Problem 4: Smallest Letter Greater Than Target

You are given an array of characters `letters` that is sorted in non-decreasing order, and a character `target`. There are at least two different characters in letters.

Write a function `next_greatest_letter()` that returns the smallest character in `letters` that is lexicographically greater than target. If such a character does not exist, return the first character in `letters`.

Lexicographic order can also be defined as alphabetic order.

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.

```python
def next_greatest_letter(letters, target):
    pass
```

Example Usage:

```python
letters = ['a', 'a', 'b', 'c', 'c', 'c', 'e', 'h', 'w']

print(next_greatest_letter(letters, 'a'))
print(next_greatest_letter(letters, 'd'))
print(next_greatest_letter(letters, 'y'))
```

Example Output:

```
b
Example 1 Explanation: The smallest character lexicographically greater than 'a' in

e
Example 2 Explanation: The smallest character lexicographically greater than 'd' in

a
Example 3 Explanation: There is no character lexicographically greater than 'y' in l
so we return letters[0]
```

## Problem 5: Find K Closest Planets

You are a starship pilot navigating the galaxy and have a list of planets, each with its distance from your current position on your star map. Given an array of planet distances `planets` sorted in ascending order and your target destination distance `target_distance`, return an array with the `k` planets that are closest to your target distance. The result should also be sorted in ascending order.

Planet with distance `a` is closer to `target_distance` than planet with distance `b` if:

- `|a - target_distance| < |b - target_distance|`
- `|a - target_distance| == |b - target_distance|` and `a < b`

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.

```python
def find_closest_planets(planets, target_distance, k):
    pass
```

Example Usage:

```python
planets1 = [100, 200, 300, 400, 500]
planets2 = [10, 20, 30, 40, 50]

print(find_closest_planets(planets1, 350, 3))
print(find_closest_planets(planets2, 25, 2))
```

Example Output:

```
[200, 300, 400]
[20, 30]
```

## Problem 6: Sorting Crystals

The Jedi Council has tasked you with organizing a collection of ancient kyber crystals. Each crystal has a unique power level represented by an integer. The kyber crystals are stored in a holocron in a completely random order, but to harness their true power, they must be arranged in ascending order based on their power levels.

Given an unsorted list of crystal power levels `crystals`, write a function that returns `crystals` as a sorted list. Your function must have `O(nlog(n))` time complexity.

```python
def sort_crystals(crystals):
    pass
```

Example Usage:

```
print(sort_crystals([5, 2, 3, 1]))
print(sort_crystals([5, 1, 1, 2, 0, 0]))
```

Example Output:

```
[1, 2, 3, 5]
[0, 0, 1, 1, 2, 5]
```

▶ 💡 **Hint: Divide and Conquer**

▶ 💡 **Hint: Merge Sort**

## Problem 7: Longest Substring With at Least K Repeating Characters

Given a string `s` and an integer `k`, use a divide and conquer approach to return the length of the longest substring of `s` such that the frequency of each character in substring is greater than or equal to `k`.

If no such substring exists, return `0`.

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 longest_substring(s, k):
    pass
```

Example Usage:

```
print(longest_substring("tatooine", 2))
print(longest_substring("chewbacca", 2))
```

Example Output:

```
2
Example 1 Explanation: The longest substring is 'oo' as 'o' is repeated 2 times.

4
Example 2 Explanation: The longest substirng is 'acca' as both 'a' and 'c' are repea
```

Close Section

## ▼ Advanced Problem Set Version 2

## Problem 1: Concert Ticket Search

You are helping a friend find a concert ticket they can afford in a sorted list `ticket_prices`. Return the index of the ticket with a price closest to, but not greater than their `budget`.

Your solution must have `O(log n)` time complexity.

```
def find_affordable_ticket(prices, budget):
    pass
```

Example Usage:

```
print(find_affordable_ticket([50, 75, 100, 150], 90))
```

Example Output:

```
1
Explantion: 75 is the closest ticket price less than or equal to 90.
It has index 1.
```

▶ 💡 **Hint: Binary Search**

▶ 💡 **Hint: `O(log n)` Time Complexity**

## Problem 2: Concert Ticket Search II

If you solved the above problem iteratively, solve it recursively. If you solved it recursively, solve it iteratively.

```
def find_affordable_ticket(prices, budget):
    pass
```

Example Usage:

```
print(find_affordable_ticket([50, 75, 100, 150], 90))
```

Example Output:

```
2
Explantion: 75 is the closest ticket price less than or equal to 90.
It has index 2.
```

## Problem 3: Organizing Setlists

You are planning a series of concerts and have a list of potential songs for the setlist, each with a specific duration. You want to create a setlist that maximizes the number of songs while ensuring that the total duration of the setlist does not exceed the time limit set for the concert.

Given an integer array `song_durations` where each element represents the duration of a song and an integer array `concert_limits` where each element represents the total time limit available for a concert, return an array `setlist_sizes` where `setlist_sizes[i]` is the maximum number of songs you can include in the playlist for concert `i` such that the total duration of the setlist is less than or equal to `concert_limits[i]`.

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 concert_playlists(song_durations, concert_limits):
    pass
```

Example Usage:

```
song_durations1 = [4, 3, 1, 2]
concert_limits1 = [5, 10, 15]

song_durations2 = [2, 3, 4, 5]
concert_limits2 = [1]

print(concert_playlists(song_durations1, concert_limits1))
print(concert_playlists(song_durations2, concert_limits2))
```

Example Output:

```
[2, 4, 4]
Example 1 Explanation:
- [3, 2] has a sum less than or equal to 5, thus 2 songs can be played at concert 1.
- [4, 3, 1, 2] has a sum less than or equal to 10, thus 4 songs can be played at con
- [4, 3, 1, 2] has a sum less than or equal to 15, thus 4 songs can be played at con

[0]
Example 2 Explanation:
- No songs are less than 1 minute long, so zero songs can be played at the concert.
```

## Problem 4: Minimum Merchandise Distribution Rate

You're in charge of distributing merchandise to different booths at a music festival, and there are `n` booths, each stocked with different amounts of merchandise. The `i` th booth has `booths[i]` items. You have `h` hours before the festival closes, and your job is to distribute all the merchandise to the attendees.

You can set a maximum distribution rate `r`, which represents the number of items you can distribute per hour. Each hour, you visit one booth and distribute `r` items from that booth. If the booth has fewer than `r` items left, you distribute all remaining items in that booth during that hour and then move on to the next hour.

Given a list of integers `booths` where each element represents the number of merchandise items at the `i` th booth, return the minimum distribution rate `r` such that you can distribute all the items within `h` hours.

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 min_distribution_rate(booths, h):
    pass
```

Example Usage:

```
print(min_distribution_rate([3, 6, 7, 11], 8))
print(min_distribution_rate([30,11,23,4,20], 5))
print(min_distribution_rate([30,11,23,4,20], 6))
```

Example Output:

```
4
30
23
```

## Problem 5: Finding the Crescendo in a Riff

You're a music producer analyzing a vocal riff in a song. The riff starts softly, builds up to a powerful high note (the crescendo), and then gradually descends. You're given an array `riff` representing the loudness of the notes in the riff. The values first increase up to the high note and then decrease.

Write a function `find_crescendo()` that returns the index of the crescendo — the highest note in the riff — using an efficient algorithm with `O(log n)` time complexity.

```
def find_crescendo(riff):
    pass
```

Example Usage:

```
print(find_crescendo([1, 3, 7, 12, 10, 6, 2]))
```

Example Output:

```
3
Explanation: The crescendo (highest note) is 12, which occurs at index 3 in the riff
```

# Problem 6: Constructing a Harmonious Sequence

You're composing a riff consisting of a sequence of musical notes. Each note is represented by an integer in the range `[1, n]`. You want to create a "harmonious" sequence that adheres to specific musical rules:

- The sequence must be a permutation of the integers from `1` to `n` (representing the notes you can use).
- For every two notes in the sequence, if you pick any three notes `note[i]`, `note[k]`, and `note[j]` such that `i < k < j`, the note at index `k` should not be exactly the midpoint between the notes at `i` and `j` (i.e., `2 * note[k]` should not equal `note[i] + note[j]`).

Given an integer `n`, return a "harmonious" sequence of notes that meets these criteria.

```
def harmonious_sequence(n):
    pass
```

Example Usage:

```
print(harmonious_sequence(4))
print(harmonious_sequence(5))
```

Example Output:

```
[1, 3, 2, 4]
Example 1 Explanation: The sequence [1, 3, 2, 4] is a harmonious sequence because it
of [1, 2, 3, 4] and satisfies the harmonious condition.

[1, 3, 5, 2, 4]
Example 2 Explanation: The sequence [1, 3, 5, 2, 4] is a harmonious sequence because
 of [1, 2, 3, 4, 5] and satisfies the harmonious condition.
```

▶ 💡 **Hint: Divide and Conquer**

# Problem 7: Longest Harmonious Subsequence

You are composing a musical piece and have a sequence of notes represented by the string `notes`. Each note in the sequence can be either in a lower octave (lowercase letter) or higher octave (uppercase letter). A sequence of notes is considered harmonious if, for every note in the sequence, both its lower and higher octave versions are present.

For example, the phrase `"aAbB"` is harmonious because both `'a'` and `'A'` appear, as well as `'b'` and `'B'`. However, the phrase `"abA"` is not harmonious because `'b'` appears, but `'B'` does not.

Given a sequence of notes `notes`, use a divide and conquer approach to return the longest harmonious subsequence within `notes`. If there are multiple, return the one that appears first. If no harmonious sequence exists, return an empty string.

```
def longest_harmonious_subsequence(notes):
    pass
```

Example Usage:

```
print(longest_harmonious_subsequence("GadaAg"))
print(longest_harmonious_subsequence("Bb"))
print(longest_harmonious_subsequence("c"))
```

Example Output:

```
aAa
Example 1 Explanation: "aAa" is a nice string because 'A/a' is the only letter of th
and both 'A' and 'a' appear. "aAa" is the longest nice substring.

Bb
Example 2 Explanation: "Bb" is a nice string because both 'B' and 'b' appear.
The whole string is a substring.

Empty String
Example 3 Explanation: There are no nice substrings.
```

Close Section