TIP102 | Intermediate Technical Interview Prep

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

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

We will approach problems using the six steps in the UMPIRE approach.

UMPIRE: Understand, Match, Plan, Implement, Review, Evaluate.

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

We will learn to:

- **Understand** the problem
- **Match** identifies common approaches you've seen/used before
- **Plan** a solution step-by-step, and
- **Implement** the solution
- **Review** your solution
- **Evaluate** your solution's time and space complexity and think critically about the advantages and disadvantages of your chosen approach.

Breakout Problems Session 2

▾ Standard Problem Set Version 1

Problem 1: Finding the Perfect Cruise

It's vacation time! Given an integer `vacation_length` and a list of integers `cruise_lengths` sorted in ascending order, use binary search to return `True` if there is a cruise length that matches `vacation_length` and `False` otherwise.

```
def find_cruise_length(cruise_lengths, vacation_length):
    pass
```

Example Usage:

```
print(find_cruise_length([9, 10, 11, 12, 13, 14, 15], 13))

print(find_cruise_length([8, 9, 12, 13, 13, 14, 15], 11))
```

Example Output:

```
True
False
```

▼ ✨ **AI Hint: Binary Search**

Key Skill: Use AI to explain code concepts

This problem requires you to understand the binary search algorithm. To learn more about this topic, check out the Binary Search section of the Unit Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the binary search algorithm. Try asking it to explain the concept first, using a real-world analogy. Once you understand the concepts, you can ask it to help you understand how to implement it in Python.

Problem 2: Booking the Perfect Cruise Cabin

As part of your cruise planning, you have a list of available cabins sorted in ascending order by their deck level. Given the list of available cabins represented by deck level, `cabins`, and an integer `preferred_deck`, write a **recursive** function `find_cabin_index()` that returns the index of `preferred_deck`. If a cabin with your `preferred_deck` does not exist in `cabins`, return the index where it would be if it were added to the list to maintain the sorted order.

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

```
def find_cabin_index(cabins, preferred_deck):
    pass
```

Example Usage:

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

Example Output:

```
2
1
4
```

▼ 💡 **Hint: Recursion**

This problem requires you to understand recursion and the differences between iteration and recursion. For reference, check out the unit cheatsheet.

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

This problem lists the constraint that the solution has `O(log n)` or logarithmic time complexity. To learn more about what that means, take a look at the Logarithmic Time Complexity section of the unit cheatsheet.

Problem 3: Count Checked In Passengers

As a cruise ship worker, you're in charge of tracking how many passengers have checked in to their rooms thus far. You are given a list of `rooms` where passengers are either checked in (represented by a `1`) or not checked in (represented by a `0`). The list is sorted, so all the `0`s appear before any `1`s.

Write a function `count_checked_in_passengers()` that efficiently counts and returns the total number of checked-in passengers (`1`s) in the list in `O(log n)` time.

```
def count_checked_in_passengers(rooms):
    pass
```

Example Usage:

```
rooms1 = [0, 0, 0, 1, 1, 1, 1]
rooms2 = [0, 0, 0, 0, 0, 1]
rooms3 = [0, 0, 0, 0, 0, 0]

print(count_checked_in_passengers(rooms1))
print(count_checked_in_passengers(rooms2))
print(count_checked_in_passengers(rooms3))
```

Example Output:

```
4
1
0
```

Problem 4: Determining Profitability of Excursions

As the activities director on a cruise ship, you're organizing excursions for the passengers. You have a sorted list of non-negative integers `excursion_counts`, where each number represents the number of passengers who have signed up for various excursions at your next cruise destination. The list is considered **profitable** if there exists a number `x` such that there are **exactly** `x` excursions that have **at least** `x` passengers signed up.

Write a function that detrmines whether `excursion_counts` is profitable. If it is profitable, return the value of `x`. If it is not profitable, return `-1`. It can be proven that if `excursion_counts` is profitable, the value for `x` is unique.

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 is_profitable(excursion_counts):
    pass
```

Example Usage:

```
print(is_profitable([3, 5]))
print(is_profitable([0, 0]))
```

Example Output:

```
2
Example 1 Explanation: There are 2 values (3 and 5) that are greater than or equal to 2.

-1
Example 2 Explanation: No numbers fit the criteria for x.
    - If x = 0, there should be 0 numbers >= x, but there are 2.
        - If x = 1, there should be 1 number >= x, but there are 0.
        - If x = 2, there should be 2 numbers >= x, but there are 0.
        - x cannot be greater since there are only 2 numbers in nums.
```

Problem 5: Finding the Shallowest Point

As the captain of the cruise ship, you need to take a detour to steer clear of an incoming storm. Given an array of integers `depths` representing the varying water depths along your potential new route, write a function `find_shallowest_point()` to help you decide whether the new route is deep enough for your ship. The function should use a divide-and-conquer approach to return the shallowest point (minimum value) in `depths`. You may not use the built-in `min()` function.

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 find_shallowest_point(arr):
    pass
```

Example Usage:

```
print(find_shallowest_point([5, 7, 2, 8, 3]))
print(find_shallowest_point([12, 15, 10, 21]))
```

Example Output:

```
2
10
```

▼ ✨ **AI Hint: Divide and Conquer**

Key Skill: Use AI to explain code concepts

Merge sort (and binary search!) are examples of algorithms that use the divide and conquer technique. To learn more about this topic, check out the Divide and Conquer and Merge Sort sections of the Unit Cheatsheet.

If you have more questions, try asking an AI tool like ChatGPT or GitHub Copilot to explain the divide and conquer technique.

You can ask it to provide a real-world analogy to help you understand the concept better. Once you grasp the idea, you can ask it to help you implement a divide and conquer algorithm in Python, such as merge sort or binary search.

Problem 6: Cruise Ship Treasure Hunt

As a fun game, the cruise ship director has organized a treasure hunt for the kids on board and hidden a chest of candy in one of the rooms on board. The rooms are organized in a `m x n` grid, where each row and each column are sorted in ascending order by room number. Given an integer representing the room number where the prize is hidden `treasure`, use a divide and conquer approach to return a tuple in the form `(row, col)` representing the row and column indices where `treasure` was found. If `treasure` is not in the matrix, return `(-1, -1)`.

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 find_treasure(matrix, treasure):
	pass
```

Example Usage:

```
rooms = [
    [1, 4, 7, 11],
    [8, 9, 10, 20],
    [11, 12, 17, 30],
    [18, 21, 23, 40]
]

print(find_treasure(rooms, 17))
print(find_treasure(rooms, 5))
```

Example Output:

```
(2, 2)
(-1, -1)
```

Close Section

▾ Standard Problem Set Version 2

Problem 1: Finding the Perfect Song

Abby Lee of Dance Moms is looking for the perfect song to choreograph a group routine to and needs a song of a specified length. Given a specific song length `length` and a list of song lengths `playlist` sorted in ascending order, use the binary search algorithm to return the index of the song in `playlist` with `length`. If no song with the target `length` exists, return `-1`.

```
def find_perfect_song(playlist, length):
        pass
```

Example Usage:

```
print(find_perfect_song([101, 102, 103, 104, 105], 103))
print(find_perfect_song([201, 202, 203, 204, 205], 206))
```

```
2
-1
```

▼ ✨ AI Hint: Binary Search

Key Skill: Use AI to explain code concepts

This problem requires you to understand the binary search algorithm. To learn more about this topic, check out the Binary Search section of the Unit Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the binary search algorithm. Try asking it to explain the concept first, using a real-world analogy. Once you understand the concepts, you can ask it to help you understand how to implement it in Python.

Problem 2: Finding Tour Dates

Your favorite artist is doing a short residency in your city and you're hoping to attend one of their concerts! But because of school, you're only free one day this month 😭. Given a sorted list of integers `tour_dates` representing the days this month your favorite artist is playing, and an integer `available` representing the day you are available, write a **recursive** function `can_attend()` that returns `True` if you will be able to attend one of their concerts (some date in `tour_dates` matches `available`) and `False` otherwise.

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

```
def can_attend(tour_dates, available):
    pass
```

Example Usage:

```
print(can_attend([1, 3, 7, 10, 12], 12))
print(can_attend([1, 3, 7, 10, 12], 5))
```

Example Output:

```
True
False
```

▼ 💡 **Hint: Recursive Helpers**

Many recursive solutions can benefit from or even require the use of helper functions. To learn more about recursive helper functions, check out the Recursive Driver and Helper Functions sections of the unit cheatsheet.

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

This problem lists the constraint that the solution has `O(log n)` or logarithmic time complexity. To learn more about what that means, take a look at the Logarithmic Time Complexity section of the unit cheatsheet.

Problem 3: Sqrt(x)

Given a non-negative integer `x`, use binary search to return the square root of `x` rounded down to the nearest integer. The returned integer should be non-negative as well.

You may not use any built-in exponent function or operator. You may not use any external libraries like `math`.

- For example, do not use `pow(x, 0.5)` or `x ** 0.5`.

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 my_sqrt(x):
    pass
```

Example Usage:

```
print(my_sqrt(4))
print(my_sqrt(8))
```

Example Output:

```
2
4
Example 2 Explanation: The square root of 8 is 2.82842..., and since we round it down
to the nearest integer, the answer is 2.
```

Problem 4: Granting Backstage Access

You're helping manage a music tour, and you have an array of integers `group_sizes` where each element represents a group of friends attending tonight's concert together. The artist has time to meet two sets of fans backstage before the show. You want to choose two groups such that the combined number of people is the highest possible while still strictly below a threshold `room_capacity`. Given the list `group_sizes` and integer `room_capacity`, use binary search to return the maximum sum of two distinct groups in `group_sizes` where the sum is less than `room_capacity`. If no such pair exists, return `-1`.

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 get_group_sum(group_sizes, room_capacity):
    pass
```

Example Usage:

```
print(get_group_sum([1,20,10,14,3,5,4,2], 12))
print(get_group_sum([10,20,30], 15))
```

Example Output:

```
11
Example 1 Explanation: We can use 1 and 10 to sum 11 which is less than 12

-1
Example 2 Explanation: In this case it is not possible to get a pair sum less than 15.
```

Problem 5: Harmonizing Two Musical Tracks

You're working as a music producer and have two tracks `track1` and `track2`, each represented by a sorted list of pitch values. Using the divide-and-conquer approach, merge the pitch values into a single, sorted sequence and return the resulting list.

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 merged_tracks(track1, track2):
    pass
```

Example Usage:

```
track1 = [1, 3, 5]
track2 = [2, 4, 6]
track3 = [10, 20]
track4 = [15, 30]

print(merged_tracks(track1, track2))
print(merged_tracks(track3, track4))
```

Example Output:

```
[1, 2, 3, 4, 5, 6]
[10, 15, 20, 30]
```

▼ ✨ AI Hint: Divide and Conquer

Key Skill: Use AI to explain code concepts

Merge sort (and binary search!) are examples of algorithms that use the divide and conquer technique. To learn more about this topic, check out the Divide and Conquer and Merge Sort sections of the Unit Cheatsheet.

If you have more questions, try asking an AI tool like ChatGPT or GitHub Copilot to explain the divide and conquer technique.

You can ask it to provide a real-world analogy to help you understand the concept better. Once you grasp the idea, you can ask it to help you implement a divide and conquer algorithm in Python, such as merge sort or binary search.

Problem 6: Merge Sort Playlist

Given a list of strings `playlist`, use merge sort to write a recursive `merge_sort_playlist()` function that accepts that returns the list of songs sorted in alphabetical order.

Pseudocode has been provided for you

```
def merge_sort_helper(left_arr, right_arr):
    # Create an empty list to store merged result list
    # Use pointers to iterate through left_arr and right_arr
        # Compare their elements, and add the smaller element to result list
        # Increment pointer of list with smaller element
    # Add any remaining elements from the left half
    # Add any remaining elements from the right half
    # Return the merged list
    pass

def merge_sort_playlist(playlist):
    # Base Case:
    # If the list has 1 or 0 elements, it's already sorted

    # Recursive Cases:
    # Divide the list into two halves
    # Merge sort first half
    # Merge sort second half
    # Use the recursive helper to merge the sorted halves (pass in sorted left half, and sor
    # Return the merged list
    pass
```

Example Usage:

```
print(merge_sort_playlist(["Formation", "Crazy in Love", "Halo"]))
print(merge_sort_playlist(["Single Ladies", "Love on Top", "Irreplaceable"]))
```

Example Output:

```
['Crazy in Love', 'Formation', 'Halo']
['Irreplaceable', 'Love on Top', 'Single Ladies']
```

▼ ✨ **AI Hint: Merge Sort**

Key Skill: Use AI to explain code concepts

This problem requires you to understand the merge sort algorithm. To learn more about this topic, check out the Merge Sort section of the Unit Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the merge sort algorithm, and break down each step of the process.

Close Section

▾ 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.

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

Example Usage:

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

Example Ouput:

```
True
False
```

▼ ✨ AI Hint: Binary Search

Key Skill: Use AI to explain code concepts

This problem requires you to understand the binary search algorithm. To learn more about this topic, check out the Binary Search section of the Unit Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the binary search algorithm. Try asking it to explain the concept first, using a real-world analogy. Once you understand the concepts, you can ask it to help you understand how to implement it in Python.

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

This problem lists the constraint that the solution has `O(log n)` or logarithmic time complexity. To learn more about what that means, take a look at the Logarithmic Time Complexity section of the unit cheatsheet. We recommend implementing your solution using binary search, then reading more about this time complexity **after** attempting this problem.

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.

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

Example Usage:

```
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.

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

Example Usage:

```
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.

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

Example Usage:

```
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 letters

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

a
Example 3 Explanation: There is no character lexicographically greater than 'y' in letters
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.

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

Example Usage:

```
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.

```
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]
```

▼ ✨ **AI Hint: Divide and Conquer**

Key Skill: Use AI to explain code concepts

Merge sort (and binary search!) are examples of algorithms that use the divide and conquer technique. To learn more about this topic, check out the Divide and Conquer and Merge Sort sections of the Unit Cheatsheet.

If you have more questions, try asking an AI tool like ChatGPT or GitHub Copilot to explain the divide and conquer technique.

You can ask it to provide a real-world analogy to help you understand the concept better. Once you grasp the idea, you can ask it to help you implement a divide and conquer algorithm in Python, such as merge sort or binary search.

▼ ✨ **AI Hint: Merge Sort**

Key Skill: Use AI to explain code concepts

This problem requires you to understand the merge sort algorithm. To learn more about this topic, check out the Merge Sort section of the Unit Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the merge sort algorithm, and break down each step of the process.

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 repeated 2 ti
```

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.
```

▼ ✨ AI Hint: Binary Search

Key Skill: Use AI to explain code concepts

This problem requires you to understand the binary search algorithm. To learn more about this topic, check out the Binary Search section of the Unit Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the binary search algorithm. Try asking it to explain the concept first, using a real-world analogy. Once you understand the concepts, you can ask it to help you understand how to implement it in Python.

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

This problem lists the constraint that the solution has `O(log n)` or logarithmic time complexity. To learn more about what that means, take a look at the Logarithmic Time Complexity section of the unit cheatsheet. We recommend implementing your solution using binary search, then reading more about this time complexity **after** attempting this problem.

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.

```python
def concert_playlists(song_durations, concert_limits):
    pass
```

Example Usage:

```python
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 concert 2.
- [4, 3, 1, 2] has a sum less than or equal to 15, thus 4 songs can be played at concert 2.

[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 is a pe
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 it is a
 of [1, 2, 3, 4, 5] and satisfies the harmonious condition.
```

▼ ✨ **AI Hint: Divide and Conquer**

Key Skill: Use AI to explain code concepts

Merge sort (and binary search!) are examples of algorithms that use the divide and conquer technique. To learn more about this topic, check out the Divide and Conquer and Merge Sort sections of the Unit Cheatsheet.

If you have more questions, try asking an AI tool like ChatGPT or GitHub Copilot to explain the divide and conquer technique.

You can ask it to provide a real-world analogy to help you understand the concept better. Once you grasp the idea, you can ask it to help you implement a divide and conquer algorithm in Python, such as merge sort or binary search.

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 the alphabe
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