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: Dictionaries

Session Overview

The focus of this session is advanced data handling in Python, focusing on functions, lists, strings, and dictionaries. Students will tackle practical programming challenges such as verifying subsequences, creating and manipulating dictionaries, and calculating values based on dynamic inputs.

The tasks are designed to enhance understanding of data structures, algorithmic thinking, and conditional logic, preparing students for more complex problem-solving scenarios involving data manipulation and retrieval.

> 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

While **each problem may call for slightly different approaches to these three steps**, the basics of the steps won't change, and it's important to engage them each time. We've built out some starting points to use in our breakout sessions, below!

Please read the following carefully (take 10 minutes as a team, if you like) and follow these basic steps, as a group, through each of the problems in your problem set.

Fun Fact: We sometimes call the main beats of problem solving, or the tasks that teams are being asked to take, our "DoNow's". If you hear a staff member using this phrase, (e.g., "Ok great! Your team is struggling with the Understand step – what might be a DoNow?") you now know what they might mean!

▼ UPI Example

Step	What is it?	Try It!
1. Understand	Here we strive to **Understand** what the interviewer is asking for. It's common to restate the problem aloud, ask questions, and consider related test cases.	• Nominate one person to share their screen so everyone is on the same page, and bring up the problem at hand. (NOTE: Please trade-off and change who is screen sharing, roughly each problem • Have one person read the problem aloud. • Have a different person restate the problem in their own words. • Have members of the group ask 2-3 questions about the problem, perhaps about the example usage, or expected output • Example: "Will the list always contain only numbers?"

2. Plan	Then we **Plan** a solution, starting with appropriate visualizations and pseudocode.	• Restate - have one person share the general idea about what the function is trying to accomplish.
		• Next, break down the problem into subproblems as a group. Each member should participate.
		  ○ If you don't know where to start, try to describe how you would solve the problem *without a computer*.
		• As a group, translate each subproblem into pseudocode.
		  ○ How do I do what I described in English in Python?
		  ○ Then, do I need to change my approach to any steps to make it work in code?

3. Implement	Now we **Implement** our solution, by translating our **Plan** into Python.	• Translate the pseudocode into Python—this is a stage at which you can consider working individually.

Breakout Problems Session 1

▾ Standard Problem Set Version 1

Problem 1: Festival Lineup

Given two lists of strings `artists` and `set_times` of length `n`, write a function `lineup()` that maps each artist to their set time.

An artist `artists[i]` has set time `set_times[i]`. Assume `i <= 0 < n` and `len(artists) == len(set_times)`.

```
def lineup(artists, set_times):
    pass
```

Example Usage:

```
artists1 = ["Kendrick Lamar", "Chappell Roan", "Mitski", "Rosalia"]
set_times1 = ["9:30 PM", "5:00 PM", "2:00 PM", "7:30 PM"]

artists2 = []
set_times2 = []

print(lineup(artists1, set_times1))
print(lineup(artists2, set_times2))
```

Example Output:

```
{"Kendrick Lamar": "9:30 PM", "Chappell Roan": "5:00 PM", "Mitski": "2:00 PM", "Rosa
{}
```

▶ 💡 **Hint: Dictionaries**

Problem 2: Planning App

You are designing an app for your festival to help attendees have the best experience possible! As part of the application, users will be able to easily search their favorite artist and find out the day, time, and stage the artist is playing at. Write a function `get_artist_info()` that accepts a string `artist` and a dictionary `festival_schedule` mapping artist's names to dictionaries containing the day, time, and stage they are playing on. Return the dictionary containing the information about the given `artist`.

If the artist searched for does not exist in `festival_schedule`, return the dictionary `{"message": "Artist not found"}`.

```
def get_artist_info(artist, festival_schedule):
    pass
```

Example Usage:

```
festival_schedule = {
    "Blood Orange": {"day": "Friday", "time": "9:00 PM", "stage": "Main Stage"},
    "Metallica": {"day": "Saturday", "time": "8:00 PM", "stage": "Main Stage"},
    "Kali Uchis": {"day": "Sunday", "time": "7:00 PM", "stage": "Second Stage"},
    "Lawrence": {"day": "Friday", "time": "6:00 PM", "stage": "Main Stage"}
}

print(get_artist_info("Blood Orange", festival_schedule))
print(get_artist_info("Taylor Swift", festival_schedule))
```

Example Output:

```
{'day': 'Friday', 'time': '9:00 PM', 'stage': 'Main Stage'}
{'message': 'Artist not found'}
```

▶ 💡 **Hint: Accessing Values in a Dictionary**

Problem 3: Ticket Sales

A dictionary `ticket_sales` is used to map ticket type to number of tickets sold. Return the total number of tickets of all types sold.

```
def total_sales(ticket_sales):
    pass
```

Example Usage:

```
ticket_sales = {"Friday": 200, "Saturday": 1000, "Sunday": 800, "3-Day Pass": 2500}

print(total_sales(ticket_sales))
```

Example Output:

```
4500
```

▶ 💡 **Hint: Accessing Keys, Values, and Key-Value Pairs**

Problem 4: Scheduling Conflict

Demand for your festival has exceeded expectations, so you're expanding the festival to span two different venues. Some artists will perform both venues, while others will perform at just one. To ensure that there are no scheduling conflicts, implement a function `identify_conflicts()` that accepts two dictionaries `venue1_schedule` and `venue2_schedule` each mapping the artists playing at the venue to their set times. Return a dictionary containing the key-value pairs that are the same in each schedule.

```
def identify_conflicts(venue1_schedule, venue2_schedule):
    pass
```

Example Usage:

```
venue1_schedule = {
    "Stromae": "9:00 PM",
    "Janelle Monáe": "8:00 PM",
    "HARDY": "7:00 PM",
    "Bruce Springsteen": "6:00 PM"
}

venue2_schedule = {
    "Stromae": "9:00 PM",
    "Janelle Monáe": "10:30 PM",
    "HARDY": "7:00 PM",
    "Wizkid": "6:00 PM"
}

print(identify_conflicts(venue1_schedule, venue2_schedule))
```

Example Output:

```
{"Stromae": "9:00 PM", "HARDY": "7:00 PM"}
```

Problem 5: Best Set

As part of the festival, attendees cast votes for their favorite set. Given a dictionary `votes` that maps attendees id numbers to the artist they voted for, return the artist that had the most number of votes. If there is a tie, return any artist with the top number of votes.

```
def best_set(votes):
    pass
```

Example Usage:

```
votes1 = {
    1234: "SZA",
    1235: "Yo-Yo Ma",
    1236: "Ethel Cain",
    1237: "Ethel Cain",
    1238: "SZA",
    1239: "SZA"
}

votes2 = {
    1234: "SZA",
    1235: "Yo-Yo Ma",
    1236: "Ethel Cain",
    1237: "Ethel Cain",
    1238: "SZA"
}

print(best_set(votes1))
print(best_set(votes2))
```

Example Output:

```
SZA
Ethel Cain
Note: SZA and Ethel Cain would both be acceptable answers for the second example
```

▶ 💡 **Hint: Frequency Maps**

Problem 6: Performances with Maximum Audience

You are given an array `audiences` consisting of positive integers representing the audience size for different performances at a music festival.

Return the combined audience size of all performances in audiences with the maximum audience size.

The audience size of a performance is the number of people who attended that performance.

```
def max_audience_performances(audiences):
    pass
```

Example Usage:

```
audiences1 = [100, 200, 200, 150, 100, 250]
audiences2 = [120, 180, 220, 150, 220]

print(max_audience_performances(audiences1))
print(max_audience_performances(audiences2))
```

Example Output:

```
250
440
```

Problem 7: Performances with Maximum Audience II

If you used a dictionary as part of your solution to `max_audience_performances()` in the previous problem, try reimplementing the function without using a dictionary. If you implemented `max_audience_performances()` without using a dictionary, try solving the problem with a dictionary.

Once you've come up with your second solution, compare the two. Is one solution better than the other? Why or why not?

```
def max_audience_performances(audiences):
    pass
```

Example Usage:

```
audiences1 = [100, 200, 200, 150, 100, 250]
audiences2 = [120, 180, 220, 150, 220]

print(max_audience_performances(audiences1))
print(max_audience_performances(audiences2))
```

Example Output:

```
250
440
```

Problem 8: Popular Song Pairs

Given an array of integers `popularity_scores` representing the popularity scores of songs in a music festival playlist, return the number of popular song pairs.

A pair `(i, j)` is called popular if the songs have the same popularity score and `i < j`.

Hint: number of pairs = (n x n-1)/2

```
def num_popular_pairs(popularity_scores):
    pass
```

Example Usage:

```
popularity_scores1 = [1, 2, 3, 1, 1, 3]
popularity_scores2 = [1, 1, 1, 1]
popularity_scores3 = [1, 2, 3]

print(num_popular_pairs(popularity_scores1))
print(num_popular_pairs(popularity_scores2))
print(num_popular_pairs(popularity_scores3))
```

Example Output:

```
4
6
0
```

▶ 💡 **Hint: Floor Division**

Problem 9: Stage Arrangement Difference Between Two Performances

You are given two strings `s` and `t` representing the stage arrangements of performers in two different performances at a music festival, such that every performer occurs at most once in `s` and `t`, and `t` is a permutation of `s`.

The stage arrangement difference between `s` and `t` is defined as the sum of the absolute difference between the index of the occurrence of each performer in `s` and the index of the occurrence of the same performer in `t`.

Return the stage arrangement difference between `s` and `t`.

A **permutation** is a rearrangement of a sequence. For example, `[3, 1, 2]` and `[2, 1 , 3]` are both permutations of the list `[1, 2, 3]`.

Hint: Absolute value function

```
def find_stage_arrangement_difference(s, t):
    """
    :type s: List[str]
    :type t: List[str]
    :rtype: int
    """
```

Example Usage:

```
s1 = ["Alice", "Bob", "Charlie"]
t1 = ["Bob", "Alice", "Charlie"]
s2 = ["Alice", "Bob", "Charlie", "David", "Eve"]
t2 = ["Eve", "David", "Bob", "Alice", "Charlie"]

print(find_stage_arrangement_difference(s1, t1))
print(find_stage_arrangement_difference(s2, t2))
```

Example Output:

```
2
12
```

▶ 💡 **Hint: Frequency Maps**

Problem 10: VIP Passes and Guests

You're given strings `vip_passes` representing the types of guests that have VIP passes, and `guests` representing the guests you have at the music festival. Each character in guests is a type of guest you have. You want to know how many of the guests you have are also VIP pass holders.

Letters are case sensitive, so "a" is considered a different type of guest from "A".

Here is the pseudocode for the problem. Implement this in Python and explain your implementation step-by-step.

```
1. Create an empty set called vip_set.
2. For each character in vip_passes, add it to vip_set.
3. Initialize a counter variable to 0.
4. For each character in guests:
   * If the character is in vip_set, increment the count by 1.
5. Return the count.
```

```
def num_VIP_guests(vip_passes, guests):
    pass
```

Example Usage:

```
vip_passes1 = "aA"
guests1 = "aAAbbbb"

vip_passes2 = "z"
guests2 = "ZZ"

print(num_VIP_guests(vip_passes1, guests1))
print(num_VIP_guests(vip_passes2, guests2))
```

Example Output:

```
3
0
```

▶ 💡 **Hint: Introduction to sets**

Problem 11: Performer Schedule Pattern

Given a string `pattern` and a string `schedule`, return `True` if `schedule` follows the same pattern. Return `False` otherwise.

Here, "follow" means a full match, such that there is a one-to-one correspondence between a letter in `pattern` and a non-empty word in `schedule`.

You are provided with a partially implemented and buggy version of the solution. Identify and fix the bugs in the code. Then, perform a thorough code review and suggest improvements.

```
def schedule_pattern(pattern, schedule):

    genres = schedule.split()

    if len(genres) == len(pattern):
        return True

    char_to_genre = {}
    genre_to_char = {}

    for char, genre in zip(pattern, genres):
        if char in char_to_genre:
            if char_to_genre[char] == genre:
                return True
        else:
            char_to_genre[char] = genre

        if genre in genre_to_char:
            if genre_to_char[genre] == char:
                return True
        else:
            genre_to_char[genre] = char

    return False
```

Example Usage:

```
pattern1 = "abba"
schedule1 = "rock jazz jazz rock"

pattern2 = "abba"
schedule2 = "rock jazz jazz blues"

pattern3 = "aaaa"
schedule3 = "rock jazz jazz rock"

print(schedule_pattern(pattern1, schedule1))
print(schedule_pattern(pattern2, schedule2))
print(schedule_pattern(pattern3, schedule3))
```

Example Output:

```
True
False
False
```

▶ 💡 **Hint: `zip()` Function**

Problem 12: Sort the Performers

You are given an array of strings `performer_names`, and an array `performance_times` that consists of distinct positive integers representing the performance durations in minutes. Both arrays are of length `n`.

For each index `i`, `performer_names[i]` and `performance_times[i]` denote the name and performance duration of the `i`th performer.

Return `performer_names` sorted in descending order by the performance durations.

```
def sort_performers(performer_names, performance_times):
    """
    :type performer_names: List[str]
    :type performance_times: List[int]
    :rtype: List[str]
    """
```

Example Usage:

```
performer_names1 = ["Mary", "John", "Emma"]
performance_times1 = [180, 165, 170]

performer_names2 = ["Alice", "Bob", "Bob"]
performance_times2 = [155, 185, 150]

print(sort_performers(performer_names1, performance_times1))
print(sort_performers(performer_names2, performance_times2))
```

Example Output:

```
["Mary", "Emma", "John"]
["Bob", "Alice", "Bob"]
```

▶ 💡 **Hint: `sorted()` Function**

Close Section

▼ Standard Problem Set Version 2

Problem 1: Space Crew

Given two lists of length `n`, `crew` and `position`, map the space station crew to their position on board the international space station.

Each crew member `crew[i]` has job `position[i]` on board, where `0 <= i < n` and `len(crew) == len(position)`.

Hint: Introduction to dictionaries

```
def space_crew(crew, position):
    pass
```

Example Usage:

```
exp70_crew = ["Andreas Mogensen", "Jasmin Moghbeli", "Satoshi Furukawa", "Loral O'Ha
exp70_positions = ["Commander", "Flight Engineer", "Flight Engineer", " Flight Engin

ax3_crew = ["Michael Lopez-Alegria", "Walter Villadei", "Alper Gezeravci", "Marcus W
ax3_positions = ["Commander", "Mission Pilot", "Mission Specialist", "Mission Specia

print(space_crew(exp70_crew, exp70_positions))
print(space_crew(ax3_crew, ax3_positions))
```

Example Output:

```
{
    "Andreas Mogensen": "Commander",
    "Jasmin Moghbeli": "Flight Engineer",
    "Satoshi Furukawa": "Flight Engineer",
    "Loral O'Hara": "Flight Engineer",
    "Konstantin Borisov": "Flight Engineer",
}

{
    "Michael López-Alegría": "Commander",
    "Walter Villadei": "Mission Pilot",
    "Alper Gezeravcı": "Mission Specialist",
    "Marcus Wandt": "Mission Specialist"
}
```

▶ 💡 **Hint: Dictionaries**

Problem 2: Space Encyclopedia

Given a dictionary `planets` that maps planet names to a dictionary containing the planet's number of moons and orbital period, write a function `planet_lookup()` that accepts a string `planet_name` and returns a string in the form `Planet <planet_name> has an orbital period of <orbital period> Earth days and has <numb`
If `planet_name` is not a key in `planets`, return `"Sorry, I have no data on that planet."`.

```
def planet_lookup(planet_name):
    pass
```

Example Usage:

```
planetary_info = {
    "Mercury": {
        "Moons": 0,
        "Orbital Period": 88
    },
    "Earth": {
        "Moons": 1,
        "Orbital Period": 365.25
    },
    "Mars": {
        "Moons": 2,
        "Orbital Period": 687
    },
    "Jupiter": {
        "Moons": 79,
        "Orbital Period": 10592
    }
}

print(planet_lookup("Jupiter"))
print(planet_lookup("Pluto"))
```

Example Output:

```
Planet Jupiter has an orbital period of 10592 Earth days and has 79 moons.
Sorry, I have no data on that planet.
```

▶ 💡 **Hint: Accessing Values in a Dictionary**

▶ 💡 **Nested Data**

Problem 3: Breathing Room

As part of your job as an astronaut, you need to perform routine safety checks. You are given a dictionary `oxygen_levels` which maps room names to current oxygen levels and two integers `min_val` and `max_val` specifying the acceptable range of oxygen levels. Return a list of room names whose values are outside the range defined by `min_val` and `max_val` (inclusive).

```
def check_oxygen_levels(oxygen_levels, min_val, max_val):
    pass
```

Example Usage:

```
oxygen_levels = {
    "Command Module": 21,
    "Habitation Module": 20,
    "Laboratory Module": 19,
    "Airlock": 22,
    "Storage Bay": 18
}

min_val = 19
max_val = 22

print(check_oxygen_levels(oxygen_levels, min_val, max_val))
```

Example Output:

```
['Storage Bay']
```

▶ 💡 **Hint: Accessing Keys, Values, and Key-Value Pairs**

Problem 4: Experiment Analysis

Write a function `data_difference()` that accepts two dictionaries `experiment1` and `experiment2` and returns a new dictionary that contains only key-value pairs found exclusively in `experiment1` but not in `experiment2`.

```
def data_difference(experiment1, experiment2):
    pass
```

Example Usage:

```
exp1_data = {'temperature': 22, 'pressure': 101.3, 'humidity': 45}
exp2_data = {'temperature': 18, 'pressure': 101.3, 'radiation': 0.5}

print(data_difference(exp1_data, exp2_data))
```

Example Output:

```
{'temperature': 22, 'humidity': 45}
```

Problem 5: Name the Node

NASA has asked the public to vote on a new name for one of the nodes in the International Space Station. Given a list of strings `votes` where each string in the list is a voter's suggested new name, implement a function `get_winner()` that returns the suggestion with the most number of votes.

If there is a tie, return either option.

```
def get_winner(votes):
    pass
```

Example Usage:

```
votes1 = ["Colbert", "Serenity", "Serenity", "Tranquility", "Colbert", "Colbert"]
votes2 = ["Colbert", "Serenity", "Serenity", "Tranquility", "Colbert"]

print(get_winner(votes1))
print(get_winner(votes2))
```

Example Output:

```
Colbert
Serenity

Note: Colbert and Serenity would both be acceptable answers for the second example
```

▶ 💡 **Hint: Frequency Maps**

Problem 6: Check if the Transmission is Complete

Ground control has sent a transmission containing important information. A complete transmission is one where every letter of the English alphabet appears at least once.

Given a string `transmission` containing only lowercase English letters, return `true` if the transmission is complete, or `false` otherwise.

```
def check_if_complete_transmission(transmission):
    """
    :type transmission: str
    :rtype: bool
    """
```

Example Usage:

```
transmission1 = "thequickbrownfoxjumpsoverthelazydog"
transmission2 = "spacetravel"

print(check_if_complete_transmission(transmission1))
print(check_if_complete_transmission(transmission2))
```

Example Output:

```
True
False
```

Problem 7: Signal Pairs

Ground control is analyzing signal patterns received from different probes. You are given a 0-indexed array `signals` consisting of distinct strings.

The string `signals[i]` can be paired with the string `signals[j]` if the string `signals[i]` is equal to the reversed string of `signals[j]`. `0 <= i < j < len(signals)`. Return the maximum number of pairs that can be formed from the array `signals`.

Note that each string can belong in at most one pair.

```
def max_number_of_string_pairs(signals):
    pass
```

Example Usage:

```
signals1 = ["cd", "ac", "dc", "ca", "zz"]
signals2 = ["ab", "ba", "cc"]
signals3 = ["aa", "ab"]

print(max_number_of_string_pairs(signals1))
print(max_number_of_string_pairs(signals2))
print(max_number_of_string_pairs(signals3))
```

Example Output:

```
2
1
0
```

Problem 8: Find the Difference of Two Signal Arrays

You are given two 0-indexed integer arrays `signals1` and `signals2`, representing signal data from two different probes. Return a list `answer` of size 2 where:

- `answer[0]` is a list of all distinct integers in `signals1` which are not present in `signals2`.
- `answer[1]` is a list of all distinct integers in `signals2` which are not present in `signals1`.

Note that the integers in the lists may be returned in any order.

Below is the pseudocode for the problem. Implement this in Python and explain your implementation step-by-step.

```
1. Convert signals1 and signals2 to sets.
2. Find the difference between set1 and set2 and store it in diff1.
3. Find the difference between set2 and set1 and store it in diff2.
4. Return the list [diff1, diff2].
```

```
def find_difference(signals1, signals2):
    pass
```

Example Usage:

```
signals1_example1 = [1, 2, 3]
signals2_example1 = [2, 4, 6]

signals1_example2 = [1, 2, 3, 3]
signals2_example2 = [1, 1, 2, 2]

print(find_difference(signals1_example1, signals2_example1))
print(find_difference(signals1_example2, signals2_example2))
```

Example Output:

```
[[1, 3], [4, 6]]
[[3], []]
```

▶ 💡 **Hint: Introduction to sets**

Problem 9: Common Signals Between Space Probes

Two space probes have collected signals represented by integer arrays `signals1` and `signals2` of sizes `n` and `m`, respectively. Calculate the following values:

- `answer1`: the number of indices `i` such that `signals1[i]` exists in `signals2`.
- `answer2`: the number of indices `j` such that `signals2[j]` exists in `signals1`.

Return `[answer1, answer2]`.

```
def find_common_signals(signals1, signals2):
    pass
```

Example Usage:

```
signals1_example1 = [2, 3, 2]
signals2_example1 = [1, 2]
print(find_common_signals(signals1_example1, signals2_example1))

signals1_example2 = [4, 3, 2, 3, 1]
signals2_example2 = [2, 2, 5, 2, 3, 6]
print(find_common_signals(signals1_example2, signals2_example2))

signals1_example3 = [3, 4, 2, 3]
signals2_example3 = [1, 5]
print(find_common_signals(signals1_example3, signals2_example3))
```

Example Output:

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

Problem 10: Common Signals Between Space Probes II

If you implemented `find_common_signals()` in the previous problem using dictionaries, try implementing `find_common_signals()` again using sets instead of dictionaries. If you implemented `find_common_signals()` using sets, use dictionaries this time.

Once you've come up with your second solution, compare the two. Is one solution better than the other? How so? Why or why not?

```
def find_common_signals(signals1, signals2):
    pass
```

Example Usage:

```
signals1_example1 = [2, 3, 2]
signals2_example1 = [1, 2]
print(find_common_signals(signals1_example1, signals2_example1))

signals1_example2 = [4, 3, 2, 3, 1]
signals2_example2 = [2, 2, 5, 2, 3, 6]
print(find_common_signals(signals1_example2, signals2_example2))

signals1_example3 = [3, 4, 2, 3]
signals2_example3 = [1, 5]
print(find_common_signals(signals1_example3, signals2_example3))
```

Example Output:

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

Problem 11: Sort Signal Data

Ground control needs to analyze the frequency of signal data received from different probes. Given an array of integers `signals`, sort the array in increasing order based on the frequency of the values. If multiple values have the same frequency, sort them in decreasing order. Return the sorted array.

Below is a buggy or incomplete version of the solution. Identify and fix the bugs in the code. Then, perform a code review and suggest improvements.

```
def frequency_sort(signals):
    freq = {}
    for signal in signals:
        if signal in freq:
            freq[signal] += 1
        else:
            freq[signal] = 0

    sorted_signals = sorted(signals, key=lambda x: (freq[x], x))

    return sorted_signals
```

Example Usage:

```
signals1 = [1, 1, 2, 2, 2, 3]
signals2 = [2, 3, 1, 3, 2]
signals3 = [-1, 1, -6, 4, 5, -6, 1, 4, 1]

print(frequency_sort(signals1))
print(frequency_sort(signals2))
print(frequency_sort(signals3))
```

Example Output:

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

▶ 💡 **Hint: `sorted()` Function**

▶ 💡 **Hint: Lambda Functions**

Problem 12: Final Communication Hub

You are given an array `paths`, where `paths[i] = [hubA, hubB]` means there exists a direct communication path going from `hubA` to `hubB`. Return the final communication hub, that is, the hub without any outgoing path to another hub.

It is guaranteed that the paths form a line without any loops, therefore, there will be exactly one final communication hub.

```
def find_final_hub(paths):
    pass
```

Example Usage:

```
paths1 = [["Earth", "Mars"], ["Mars", "Titan"], ["Titan", "Europa"]]
paths2 = [["Alpha", "Beta"], ["Gamma", "Alpha"], ["Beta", "Delta"]]
paths3 = [["StationA", "StationZ"]]

print(find_final_hub(paths1))
print(find_final_hub(paths2))
print(find_final_hub(paths3))
```

Example Output:

```
"Europa"
"Delta"
"StationZ"
```

Close Section

▸ Advanced Problem Set Version 1

▸ Advanced Problem Set Version 2