# TIP102 | Intermediate Technical Interview Prep

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

👋 **IMPORTANT:** This session will take place on **Thursday, June 26th at 3:00PM PDT**.

# Session 1: Review

## Session Overview

In this unit, we will transition from the UPI method to the full UMPIRE method. Students will review content from Units 1-3 by matching each problem to a data structure and/or strategy introduced in previous units before solving. Students will also practice evaluating the time and space complexity of each of problem. Problems will cover strings, arrays, hash tables (dictionaries), stacks, queues, and the two pointer technique.

> 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<br>• Have one person read the problem aloud.<br>• Have a different person restate the problem in their own words.<br>• Have members of the group ask 2-3 questions about the problem, perhaps about the example usage, or expected output<br>• 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.<br>• Next, break down the problem into subproblems as a group. Each member should participate.<br>  ○ If you don't know where to start, try to describe how you would solve the problem *without a computer*.<br>• As a group, translate each subproblem into pseudocode.<br>  ○ How do I do what I described in English in Python?<br>  ○ 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: NFT Name Extractor

You're curating a large collection of NFTs for a digital art gallery, and your first task is to extract the names of these NFTs from a given list of dictionaries. Each dictionary in the list represents an NFT, and contains information such as the name, creator, and current value.

Write the `extract_nft_names()` function, which takes in this list and returns a list of all NFT names.

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 extract_nft_names(nft_collection):
    pass
```

Example Usage:

```
def extract_nft_names(nft_collection):
    nft_names = []
    for nft in nft_collection:
        nft_names.append(nft["name"])
    return nft_names

# Example usage:
nft_collection = [
    {"name": "Abstract Horizon", "creator": "ArtByAlex", "value": 5.4},
    {"name": "Pixel Dreams", "creator": "DreamyPixel", "value": 7.2},
    {"name": "Future City", "creator": "UrbanArt", "value": 3.8}
]

nft_collection_2 = [
    {"name": "Crypto Kitty", "creator": "CryptoPets", "value": 10.5},
    {"name": "Galactic Voyage", "creator": "SpaceArt", "value": 6.7}
]

nft_collection_3 = [
    {"name": "Golden Hour", "creator": "SunsetArtist", "value": 8.9}
]

print(extract_nft_names(nft_collection))
print(extract_nft_names(nft_collection_2))
print(extract_nft_names(nft_collection_3))
```

Example Output:

```
['Abstract Horizon', 'Pixel Dreams', 'Future City']
['Crypto Kitty', 'Galactic Voyage']
['Golden Hour']
```

### ▼ 💡 Hint: Big O (Time & Space Complexity)

Big O notation is a mathematical notation in computer science used to describe the the time and space complexity of an algorithm. Time complexity is the amount of time an algorithm or function takes to run in comparison to the size of the input data. Space complexity is the amount of extra memory or space an algorithm or function needs to complete its task in comparison to the size of the input data.

For your convenience, we've included a summary of the three most common Big O functions below.

Common Big O includes:

- **O(1) - Constant Time** No matter the size of your input data, the function takes a fixed amount of time or memory to complete its task.

  Example: Summing two numbers

```
def sum(a, b):
    return a + b
```

It takes the computer roughly the same amount of time to sum `a` and `b` no matter how large the two numbers are.

- **O(n) - Linear Time** The amount of time or memory your function needs grows linearly with the size of your input data.

  Example: Printing each item in a list

  ```
  def print_list(lst):
      for item in lst:
          print(item)
  ```

  The computer has to perform one extra print statement for each extra item there is in the list, so the length of time it takes to print the list will be proportional to the number of items in the list. We expect that it will take 1000 times longer to print a list with 1000 elements than it will to print a list with just 1 element.

- **O(n²) - Quadratic Time** The amount of time or memory your function needs grows quadratically with the size of your input data.

  Example: Finding Duplicates Using a Nested For Loop

  ```
  def find_duplicates(lst):
      n = len(lst)
      for i in range(n):
          for j in range(i + 1, n):
              if lst[i] == lst[j]:
                  print(f"Duplicate found: {lst[i]}")
                  return True
      return False
  ```

  The function compares each element in the list to every other element in the list, which means we perform roughly n² comparisons where n is the length of our input list `lst`, so it will take n² time to complete all comparisons. We can expect that for a list of size 2, we will perform roughly 4 comparisons whereas for a list of size 10 we will perform roughly 100 comparisons.

## ▼ ✨ AI Hint: Decoding Big O

*Key Skill: Use AI to explain code concepts*

Big O is a big topic, and kind of tricky to wrap your head around! If you're feeling confused, try asking an AI tool like ChatGPT or GitHub Copilot to explain it to you:

*"You're an expert computer science tutor for a Python-based technical interviewing prep course. Can you use an analogy to help me understand Big O notation? Please explain the concept of time and space complexity in a way that is easy to understand."*

Once it gives you an answer, you can ask follow-up questions to clarify any points that are still confusing. Be patient with yourself, and remember that this is a complex topic that takes time to fully understand!

## Problem 2: NFT Collection Review

You're responsible for ensuring the quality of the NFT collection before it is displayed in the virtual gallery. One of your tasks is to review and debug the code that extracts the names of NFTs from the collection. A junior developer wrote the initial version of this function, but it contains some bugs that prevent it from working correctly.

**Task:**

1. Review the provided code and identify the bug(s).
2. Explain what the bug is and how it affects the output.
3. Refactor the code to fix the bug(s) and provide the correct implementation.

```
def extract_nft_names(nft_collection):
    nft_names = []
    for nft in nft_collection:
        nft_names += nft["name"]
    return nft_names
```

Example Usage:

```
nft_collection = [
    {"name": "Abstract Horizon", "creator": "ArtByAlex", "value": 5.4},
    {"name": "Pixel Dreams", "creator": "DreamyPixel", "value": 7.2}
]

nft_collection_2 = [
    {"name": "Golden Hour", "creator": "SunsetArtist", "value": 8.9}
]

nft_collection_3 = []

print(extract_nft_names(nft_collection))
print(extract_nft_names(nft_collection_2))
print(extract_nft_names(nft_collection_3))
```

Example Output:

```
['Abstract Horizon', 'Pixel Dreams']
['Golden Hour']
[]
```

## Problem 3: Identify Popular Creators

You have been tasked with identifying the most popular NFT creators in your collection. A creator is considered "popular" if they have created more than one NFT in the collection.

Write the `identify_popular_creators()` function, which takes a list of NFTs and returns a list of the names of popular creators.

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 identify_popular_creators(nft_collection):
    pass
```

Example Usage:

```
nft_collection = [
    {"name": "Abstract Horizon", "creator": "ArtByAlex", "value": 5.4},
    {"name": "Pixel Dreams", "creator": "DreamyPixel", "value": 7.2},
    {"name": "Urban Jungle", "creator": "ArtByAlex", "value": 4.5}
]

nft_collection_2 = [
    {"name": "Crypto Kitty", "creator": "CryptoPets", "value": 10.5},
    {"name": "Galactic Voyage", "creator": "SpaceArt", "value": 6.7},
    {"name": "Future Galaxy", "creator": "SpaceArt", "value": 8.3}
]

nft_collection_3 = [
    {"name": "Golden Hour", "creator": "SunsetArtist", "value": 8.9}
]

print(identify_popular_creators(nft_collection))
print(identify_popular_creators(nft_collection_2))
print(identify_popular_creators(nft_collection_3))
```

Example Output:

```
['ArtByAlex']
['SpaceArt']
[]
```

# Problem 4: NFT Collection Statistics

You want to provide an overview of the NFT collection to potential buyers. One key statistic is the average value of the NFTs in the collection. However, if the collection is empty, the average value should be reported as `0`.

Write the `average_nft_value` function, which calculates and returns the average value of the NFTs in the collection.

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 average_nft_value(nft_collection):
    pass
```

Example Usage:

```
nft_collection = [
    {"name": "Abstract Horizon", "creator": "ArtByAlex", "value": 5.4},
    {"name": "Pixel Dreams", "creator": "DreamyPixel", "value": 7.2},
    {"name": "Urban Jungle", "creator": "ArtByAlex", "value": 4.5}
]
print(average_nft_value(nft_collection))

nft_collection_2 = [
    {"name": "Golden Hour", "creator": "SunsetArtist", "value": 8.9},
    {"name": "Sunset Serenade", "creator": "SunsetArtist", "value": 9.4}
]
print(average_nft_value(nft_collection_2))

nft_collection_3 = []
print(average_nft_value(nft_collection_3))
```

Example Output:

```
5.7
9.15
0
```

# Problem 5: NFT Tag Search

Some NFTs are grouped into collections, and each collection might contain multiple NFTs. Additionally, each NFT can have a list of tags describing its style or theme (e.g., `"abstract"`, `"landscape"`, `"modern"`). You need to search through these nested collections to find all NFTs that contain a specific tag.

Write the `search_nft_by_tag()` function, which takes in a nested list of NFT collections and a tag to search for. The function should return a list of NFT names that have the specified tag.

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 search_nft_by_tag(nft_collections, tag):
    pass
```

Example Usage:

```
nft_collections = [
    [
        {"name": "Abstract Horizon", "tags": ["abstract", "modern"]},
        {"name": "Pixel Dreams", "tags": ["pixel", "retro"]}
    ],
    [
        {"name": "Urban Jungle", "tags": ["urban", "landscape"]},
        {"name": "City Lights", "tags": ["modern", "landscape"]}
    ]
]

nft_collections_2 = [
    [
        {"name": "Golden Hour", "tags": ["sunset", "landscape"]},
        {"name": "Sunset Serenade", "tags": ["sunset", "serene"]}
    ],
    [
        {"name": "Pixel Odyssey", "tags": ["pixel", "adventure"]}
    ]
]

nft_collections_3 = [
    [
        {"name": "The Last Piece", "tags": ["finale", "abstract"]}
    ],
    [
        {"name": "Ocean Waves", "tags": ["seascape", "calm"]},
        {"name": "Mountain Peak", "tags": ["landscape", "adventure"]}
    ]
]

print(search_nft_by_tag(nft_collections, "landscape"))
print(search_nft_by_tag(nft_collections_2, "sunset"))
print(search_nft_by_tag(nft_collections_3, "modern"))
```

Example Output:

```
['Urban Jungle', 'City Lights']
['Golden Hour', 'Sunset Serenade']
[]
```

# Problem 6: NFT Queue Processing

NFTs are added to a processing queue before they are displayed. The queue processes NFTs in a First-In, First-Out (FIFO) manner. Each NFT has a processing time, and you need to determine the order in which NFTs should be processed based on their initial position in the queue.

Write the `process_nft_queue()` function, which takes a list of NFTs. The function should return a list of NFT names in the order they were processed.

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 process_nft_queue(nft_queue):
    pass
```

Example Usage:

```
nft_queue = [
    {"name": "Abstract Horizon", "processing_time": 2},
    {"name": "Pixel Dreams", "processing_time": 3},
    {"name": "Urban Jungle", "processing_time": 1}
]
print(process_nft_queue(nft_queue))

nft_queue_2 = [
    {"name": "Golden Hour", "processing_time": 4},
    {"name": "Sunset Serenade", "processing_time": 2},
    {"name": "Ocean Waves", "processing_time": 3}
]
print(process_nft_queue(nft_queue_2))

nft_queue_3 = [
    {"name": "Crypto Kitty", "processing_time": 5},
    {"name": "Galactic Voyage", "processing_time": 6}
]
print(process_nft_queue(nft_queue_3))
```

Example Output:

```
['Abstract Horizon', 'Pixel Dreams', 'Urban Jungle']
['Golden Hour', 'Sunset Serenade', 'Ocean Waves']
['Crypto Kitty', 'Galactic Voyage']
```

# Problem 7: Validate NFT Addition

You want to ensure that NFTs are added in a balanced way. For example, every `"add"` action must be properly closed by a corresponding `"remove"` action.

Write the `validate_nft_actions()` function, which takes a list of actions (either `"add"` or `"remove"`) and returns `True` if the actions are balanced, and `False` otherwise.

A sequence of actions is considered balanced if every `"add"` has a corresponding `"remove"` and no `"remove"` occurs before an `"add"`.

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 validate_nft_actions(actions):
    pass
```

Example Usage:

```
actions = ["add", "add", "remove", "remove"]
actions_2 = ["add", "remove", "add", "remove"]
actions_3 = ["add", "remove", "remove", "add"]

print(validate_nft_actions(actions))
print(validate_nft_actions(actions_2))
print(validate_nft_actions(actions_3))
```

Example Output:

```
True
True
False
```

## Problem 8: Find Closest NFT Values

Buyers often look for NFTs that are closest in value to their budget. Given a sorted list of NFT values and a budget, you need to find the two NFT values that are closest to the given budget: one that is just below or equal to the budget and one that is just above or equal to the budget. If an exact match exists, it should be included as one of the values.

Write the `find_closest_nft_values()` function, which takes a sorted list of NFT values and a budget, and returns the pair of the two closest NFT values.

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_nft_values(nft_values, budget):
    pass
```

Example Usage:

```
nft_values = [3.5, 5.4, 7.2, 9.0, 10.5]
nft_values_2 = [2.0, 4.5, 6.3, 7.8, 12.1]
nft_values_3 = [1.0, 2.5, 4.0, 6.0, 9.0]

print(find_closest_nft_values(nft_values, 8.0))
print(find_closest_nft_values(nft_values_2, 6.5))
print(find_closest_nft_values(nft_values_3, 3.0))
```

Example Output:

```
(7.2, 9.0)
(6.3, 7.8)
(2.5, 4.0)
```

Close Section

## ▼ Standard Problem Set Version 2

# Problem 1: Meme Length Filter

You need to filter out memes that are too long from your dataset. Memes that exceed a certain length are less likely to go viral.

Write the `filter_meme_lengths()` function, which filters out memes whose lengths exceed a given limit. The function should return a list of meme texts that are within the acceptable length.

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 filter_meme_lengths(memes, max_length):
    pass
```

Example Usage:

```
memes = ["This is hilarious!", "A very long meme that goes on and on and on...", "Short and
memes_2 = ["Just right", "This one's too long though, sadly", "Perfect length", "A bit too w
memes_3 = ["Short", "Tiny meme", "Small but impactful", "Extremely lengthy meme that no one

print(filter_meme_lengths(memes, 20))
print(filter_meme_lengths(memes_2, 15))
print(filter_meme_lengths(memes_3, 10))
```

Example Output:

```
['This is hilarious!', 'Short and sweet']
['Just right', 'Perfect length']
['Short', 'Tiny meme']
```

### ▼ 💡 Hint: Big O (Time & Space Complexity)

Big O notation is a mathematical notation in computer science used to describe the the time and space complexity of an algorithm. Time complexity is the amount of time an algorithm or function takes to run in comparison to the size of the input data. Space complexity is the amount of extra memory or space an algorithm or function needs to complete its task in comparison to the size of the input data.

For your convenience, we've included a summary of the three most common Big O functions below.

Common Big O includes:

- **O(1) - Constant Time** No matter the size of your input data, the function takes a fixed amount of time or memory to complete its task.

  Example: Summing two numbers

  ```
  def sum(a, b):
      return a + b
  ```

  It takes the computer roughly the same amount of time to sum `a` and `b` no matter how large the two numbers are.

- **O(n) - Linear Time** The amount of time or memory your function needs grows linearly with the size of your input data.

  Example: Printing each item in a list

  ```
  def print_list(lst):
      for item in lst:
          print(item)
  ```

  The computer has to perform one extra print statement for each extra item there is in the list, so the length of time it takes to print the list will be proportional to the number of items in the list. We expect that it will take 1000 times longer to print a list with 1000 elements than it will to print a list with just 1 element.

- **O(n²) - Quadratic Time** The amount of time or memory your function needs grows quadratically with the size of your input data.

  Example: Finding Duplicates Using a Nested For Loop

  ```
  def find_duplicates(lst):
      n = len(lst)
      for i in range(n):
          for j in range(i + 1, n):
              if lst[i] == lst[j]:
                  print(f"Duplicate found: {lst[i]}")
                  return True
      return False
  ```

  The function compares each element in the list to every other element in the list, which means we perform roughly n² comparisons where n is the length of our input list `lst`, so it will take n² time to complete all comparisons. We can expect that for a list of size 2, we will perform roughly 4 comparisons whereas for a list of size 10 we will perform roughly 100 comparisons.

▼ ✨ AI Hint: Decoding Big O

*Key Skill: Use AI to explain code concepts*

Big O is a big topic, and kind of tricky to wrap your head around! If you're feeling confused, try asking an AI tool like ChatGPT or GitHub Copilot to explain it to you:

*"You're an expert computer science tutor for a Python-based technical interviewing prep course. Can you use an analogy to help me understand Big O notation? Please explain the concept of time and space complexity in a way that is easy to understand."*

Once it gives you an answer, you can ask follow-up questions to clarify any points that are still confusing. Be patient with yourself, and remember that this is a complex topic that takes time to fully understand!

## Problem 2: Top Meme Creators

You want to identify the top meme creators based on the number of memes they have created.

Write the `count_meme_creators()` function, which takes a list of meme dictionaries and returns the creators' names and the number of memes they have created.

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 count_meme_creators(memes):
    pass
```

Example Usage:

```
memes = [
    {"creator": "Alex", "text": "Meme 1"},
    {"creator": "Jordan", "text": "Meme 2"},
    {"creator": "Alex", "text": "Meme 3"},
    {"creator": "Chris", "text": "Meme 4"},
    {"creator": "Jordan", "text": "Meme 5"}
]

memes_2 = [
    {"creator": "Sam", "text": "Meme 1"},
    {"creator": "Sam", "text": "Meme 2"},
    {"creator": "Sam", "text": "Meme 3"},
    {"creator": "Taylor", "text": "Meme 4"}
]

memes_3 = [
    {"creator": "Blake", "text": "Meme 1"},
    {"creator": "Blake", "text": "Meme 2"}
]

print(count_meme_creators(memes))
print(count_meme_creators(memes_2))
print(count_meme_creators(memes_3))
```

Example Output:

```
{'Alex': 2, 'Jordan': 2, 'Chris': 1}
{'Sam': 3, 'Taylor': 1}
{'Blake': 2}
```

## Problem 3: Meme Trend Identification

You're tasked with identifying trending memes. A meme is considered "trending" if it appears in the dataset multiple times.

Write the `find_trending_memes()` function, which takes a list of meme texts and returns a list of trending memes, where a trending meme is defined as a meme that appears more than once in the 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 find_trending_memes(memes):
    pass
```

Example Usage:

```
memes = ["Dogecoin to the moon!", "One does not simply walk into Mordor", "Dogecoin to the m
memes_2 = ["Surprised Pikachu", "Expanding brain", "This is fine", "Surprised Pikachu", "Sur
memes_3 = ["Y U No?", "First world problems", "Philosoraptor", "Bad Luck Brian"]

print(find_trending_memes(memes))
print(find_trending_memes(memes_2))
print(find_trending_memes(memes_3))
```

Example Output:

```
['Dogecoin to the moon!', 'One does not simply walk into Mordor']
['Surprised Pikachu']
[]
```

## Problem 4: Reverse Meme Order

You want to see how memes would trend if they were posted in reverse order.

Write the `reverse_memes()` function, which takes a list of memes (representing the order they were posted) and returns a new list with the memes in reverse 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 reverse_memes(memes):
    pass
```

Example Usage:

```
memes = ["Dogecoin to the moon!", "Distracted boyfriend", "One does not simply walk into Mor
memes_2 = ["Surprised Pikachu", "Expanding brain", "This is fine"]
memes_3 = ["Y U No?", "First world problems", "Philosoraptor", "Bad Luck Brian"]

print(reverse_memes(memes))
print(reverse_memes(memes_2))
print(reverse_memes(memes_3))
```

Example Output:

```
['One does not simply walk into Mordor', 'Distracted boyfriend', 'Dogecoin to the moon!']
['This is fine', 'Expanding brain', 'Surprised Pikachu']
['Bad Luck Brian', 'Philosoraptor', 'First world problems', 'Y U No?']
```

# Problem 5: Trending Meme Pairs

You've been given partially completed code to identify pairs of memes that frequently appear together in posts. However, before you can complete the implementation, you need to ensure the plan is correct and then review the provided code to identify and fix any potential issues.

Your task is to:

1. **Plan:**

   Write a detailed plan (pseudocode or step-by-step instructions) on how you would approach solving this problem. Consider how you would:

   - Iterate through each post.
   - Generate pairs of memes.
   - Count the frequency of each pair.
   - Identify pairs that appear more than once.
   - Ensure the final result is accurate and efficient.

2. **Review:**

   Examine the provided code and answer the following questions:

   - Are there any logical errors in the code? If so, what are they, and how would you fix them?
   - Are there any inefficiencies in the code that could be improved? If so, how would you optimize it?
   - Does the code correctly handle edge cases, such as an empty list of posts or posts with only one meme?

```
def find_trending_meme_pairs(meme_posts):
    pair_count = {}

    for post in meme_posts:
        for i in range(len(post)):
            for j in range(len(post)):
                if i != j:
                    meme1 = post[i]
                    meme2 = post[j]

                    if meme1 < meme2:
                        meme1, meme2 = meme2, meme1
                    pair = (meme1, meme2)
                    if pair in pair_count:
                        pair_count[pair] += 1
                    else:
                        pair_count[pair] = 1

    trending_pairs = []
    for pair in pair_count:
        if pair_count[pair] >= 2:
            trending_pairs.append(pair)

    return trending_pairs
```

Example Usage:

```
meme_posts_1 = [
    ["Dogecoin to the moon!", "Distracted boyfriend"],
    ["One does not simply walk into Mordor", "Dogecoin to the moon!"],
    ["Dogecoin to the moon!", "Distracted boyfriend", "One does not simply walk into Mordor"
    ["Distracted boyfriend", "One does not simply walk into Mordor"]
]

meme_posts_2 = [
    ["Surprised Pikachu", "This is fine"],
    ["Expanding brain", "Surprised Pikachu"],
    ["This is fine", "Expanding brain"],
    ["Surprised Pikachu", "This is fine"]
]

meme_posts_3 = [
    ["Y U No?", "First world problems"],
    ["Philosoraptor", "Bad Luck Brian"],
    ["First world problems", "Philosoraptor"],
    ["Y U No?", "First world problems"]
]

print(find_trending_meme_pairs(meme_posts))
print(find_trending_meme_pairs(meme_posts_2))
print(find_trending_meme_pairs(meme_posts_3))
```

Example Output:

```
[('Distracted boyfriend', 'Dogecoin to the moon!'), ('Dogecoin to the moon!', 'One does not s
[('Surprised Pikachu', 'This is fine')]
[('First world problems', 'Y U No?')]
```

## Problem 6: Meme Popularity Queue

You're tasked with analyzing the order in which memes gain popularity. Memes are posted in a sequence, and their popularity grows as they are reposted.

Write the `simulate_meme_reposts()` function, which takes a list of memes (representing their initial posting order) and simulate their reposting by processing each meme in the queue. Each meme can be reposted multiple times, and for each repost, it should be added back to the queue. The function should return the final order in which all reposts are processed.

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 simulate_meme_reposts(memes, reposts):
    pass
```

Example Usage:

```
memes = ["Distracted boyfriend", "Dogecoin to the moon!", "One does not simply walk into Mor
reposts = [2, 1, 3]

memes_2 = ["Surprised Pikachu", "This is fine", "Expanding brain"]
reposts = [1, 2, 2]

memes_3 = ["Y U No?", "Philosoraptor"]
reposts = [3, 1]

print(simulate_meme_reposts(memes, reposts))
print(simulate_meme_reposts(memes_2, reposts))
print(simulate_meme_reposts(memes_3, reposts))
```

Example Output:

```
['Distracted boyfriend', 'Dogecoin to the moon!', 'One does not simply walk into Mordor', 'D
['Surprised Pikachu', 'This is fine', 'Expanding brain', 'This is fine', 'Expanding brain']
['Y U No?', 'Philosoraptor', 'Y U No?', 'Y U No?']
```

# Problem 7: Search for Viral Meme Groups

You're interested in identifying groups of memes that, when combined, have a total popularity score closest to a target value. Each meme has an associated popularity score, and you want to find the two memes whose combined popularity score is closest to the target value. The list of memes is already sorted by their popularity scores.

Write the `find_closest_meme_pair()` function, which takes a sorted list of memes (each with a name and a popularity score) and a target popularity score. The function should return the names of the two memes whose combined popularity score is closest to the target.

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_meme_pair(memes, target):
    pass
```

Example Usage:

```
memes_1 = [("Distracted boyfriend", 5), ("Dogecoin to the moon!", 7), ("One does not simply
memes_2 = [("Surprised Pikachu", 2), ("This is fine", 6), ("Expanding brain", 9), ("Y U No?"
memes_3 = [("Philosoraptor", 1), ("Bad Luck Brian", 4), ("First world problems", 8), ("Y U N

print(find_closest_meme_pair(memes_1, 13))
print(find_closest_meme_pair(memes_2, 10))
print(find_closest_meme_pair(memes_3, 12))
```

Example Output:

```
('Distracted boyfriend', 'Dogecoin to the moon!')
('Surprised Pikachu', 'Expanding brain')
('Bad Luck Brian', 'First world problems')
```

# Problem 8: Analyze Meme Trends

You need to analyze the trends of various memes over time. You have a dataset where each meme has a name, a list of daily popularity scores (number of reposts each day), and other metadata.

Write the `find_trending_meme()` function, which takes in a list of memes (each with a name and a list of daily repost counts) and a time range (represented by a start and end day, inclusive). The function should return the name of the meme with the highest average reposts over the specified period. If there is a tie, return the meme that appears first in the 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 find_trending_meme(memes, start_day, end_day):
    pass
```

Example Usage:

```
memes = [
    {"name": "Distracted boyfriend", "reposts": [5, 3, 2, 7, 6]},
    {"name": "Dogecoin to the moon!", "reposts": [2, 4, 6, 8, 10]},
    {"name": "One does not simply walk into Mordor", "reposts": [3, 3, 5, 4, 2]}
]

memes_2 = [
    {"name": "Surprised Pikachu", "reposts": [2, 1, 4, 5, 3]},
    {"name": "This is fine", "reposts": [3, 5, 2, 6, 4]},
    {"name": "Expanding brain", "reposts": [4, 2, 1, 4, 2]}
]

memes_3 = [
    {"name": "Y U No?", "reposts": [1, 2, 1, 2, 1]},
    {"name": "Philosoraptor", "reposts": [3, 1, 3, 1, 3]}
]
print(find_trending_meme(memes, 1, 3))
print(find_trending_meme(memes_2, 0, 2))
print(find_trending_meme(memes_3, 2, 4))
```

Example Output:

```
Dogecoin to the moon!
This is fine
Philosoraptor
```

Close Section

## ▾ Advanced Problem Set Version 1

# Problem 1: Brand Filter

You're tasked with filtering out brands that are not sustainable from a list of fashion brands. A sustainable brand is defined as one that meets a specific criterion, such as using eco-friendly materials, ethical labor practices, or being carbon-neutral.

Write the `filter_sustainable_brands()` function, which takes a list of brands and a criterion, then returns a list of brands that meet the criterion.

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 filter_sustainable_brands(brands, criterion):
    pass
```

Example Usage:

```
brands = [
    {"name": "EcoWear", "criteria": ["eco-friendly", "ethical labor"]},
    {"name": "FastFashion", "criteria": ["cheap materials", "fast production"]},
    {"name": "GreenThreads", "criteria": ["eco-friendly", "carbon-neutral"]},
    {"name": "TrendyStyle", "criteria": ["trendy designs"]}
]

brands_2 = [
    {"name": "Earthly", "criteria": ["ethical labor", "fair wages"]},
    {"name": "FastStyle", "criteria": ["mass production"]},
    {"name": "NatureWear", "criteria": ["eco-friendly"]},
    {"name": "GreenFit", "criteria": ["recycled materials", "eco-friendly"]}
]

brands_3 = [
    {"name": "OrganicThreads", "criteria": ["organic cotton", "fair trade"]},
    {"name": "GreenLife", "criteria": ["recycled materials", "carbon-neutral"]},
    {"name": "FastCloth", "criteria": ["cheap production"]}
]

print(filter_sustainable_brands(brands, "eco-friendly"))
print(filter_sustainable_brands(brands_2, "ethical labor"))
print(filter_sustainable_brands(brands_3, "carbon-neutral"))
```

Example Output:

```
['EcoWear', 'GreenThreads']
['Earthly']
['GreenLife']
```

▼ 💡 **Hint: Big O (Time & Space Complexity)**

Big O notation is a mathematical notation in computer science used to describe the the time and space complexity of an algorithm. Time complexity is the amount of time an algorithm or function takes to run in comparison to the size of the input data. Space complexity is the amount of extra memory or space an algorithm or function needs to complete its task in comparison to the size of the input data.

For your convenience, we've included a summary of the three most common Big O functions below.

Common Big O includes:

- **O(1) - Constant Time** No matter the size of your input data, the function takes a fixed amount of time or memory to complete its task.

  Example: Summing two numbers

  ```
  def sum(a, b):
      return a + b
  ```

It takes the computer roughly the same amount of time to sum `a` and `b` no matter how large the two numbers are.

- **O(n) - Linear Time** The amount of time or memory your function needs grows linearly with the size of your input data.

  Example: Printing each item in a list

  ```
  def print_list(lst):
      for item in lst:
          print(item)
  ```

  The computer has to perform one extra print statement for each extra item there is in the list, so the length of time it takes to print the list will be proportional to the number of items in the list. We expect that it will take 1000 times longer to print a list with 1000 elements than it will to print a list with just 1 element.

- **O(n²) - Quadratic Time** The amount of time or memory your function needs grows quadratically with the size of your input data.

  Example: Finding Duplicates Using a Nested For Loop

  ```
  def find_duplicates(lst):
      n = len(lst)
      for i in range(n):
          for j in range(i + 1, n):
              if lst[i] == lst[j]:
                  print(f"Duplicate found: {lst[i]}")
                  return True
      return False
  ```

  The function compares each element in the list to every other element in the list, which means we perform roughly n² comparisons where n is the length of our input list `lst`, so it will take n² time to complete all comparisons. We can expect that for a list of size 2, we will perform roughly 4 comparisons whereas for a list of size 10 we will perform roughly 100 comparisons.

### ▼ ✨ AI Hint: Decoding Big O

*Key Skill: Use AI to explain code concepts*

Big O is a big topic, and kind of tricky to wrap your head around! If you're feeling confused, try asking an AI tool like ChatGPT or GitHub Copilot to explain it to you:

*"You're an expert computer science tutor for a Python-based technical interviewing prep course. Can you use an analogy to help me understand Big O notation? Please explain the concept of time and space complexity in a way that is easy to understand."*

Once it gives you an answer, you can ask follow-up questions to clarify any points that are still confusing. Be patient with yourself, and remember that this is a complex topic that takes time to fully understand!

## Problem 2: Eco-Friendly Materials

Certain materials are recognized as eco-friendly due to their low environmental impact. You need to track which materials are used by various brands and count how many times each material appears across all brands. This will help identify the most commonly used eco-friendly materials.

Write the `count_material_usage()` function, which takes a list of brands (each with a list of materials) and returns the material names and the number of times each material appears across all brands.

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 count_material_usage(brands):
    pass
```

Example Usage:

```
brands = [
    {"name": "EcoWear", "materials": ["organic cotton", "recycled polyester"]},
    {"name": "GreenThreads", "materials": ["organic cotton", "bamboo"]},
    {"name": "SustainableStyle", "materials": ["bamboo", "recycled polyester"]}
]

brands_2 = [
    {"name": "NatureWear", "materials": ["hemp", "linen"]},
    {"name": "Earthly", "materials": ["organic cotton", "hemp"]},
    {"name": "GreenFit", "materials": ["linen", "recycled wool"]}
]

brands_3 = [
    {"name": "OrganicThreads", "materials": ["organic cotton"]},
    {"name": "EcoFashion", "materials": ["recycled polyester", "hemp"]},
    {"name": "GreenLife", "materials": ["recycled polyester", "bamboo"]}
]

print(count_material_usage(brands))
print(count_material_usage(brands_2))
print(count_material_usage(brands_3))
```

Example Output:

```
{'organic cotton': 2, 'recycled polyester': 2, 'bamboo': 2}
{'hemp': 2, 'linen': 2, 'organic cotton': 1, 'recycled wool': 1}
{'organic cotton': 1, 'recycled polyester': 2, 'hemp': 1, 'bamboo': 1}
```

## Problem 3: Fashion Trends

In the fast-changing world of fashion, certain materials and practices become trending based on how frequently they are adopted by brands. You want to identify which materials and practices are trending. A material or practice is considered "trending" if it appears in the dataset more than once.

Write the `find_trending_materials()` function, which takes a list of brands (each with a list of materials or practices) and returns a list of materials or practices that are trending (i.e., those that appear more than once across all brands).

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_trending_materials(brands):
    pass
```

Example Usage:

```
brands = [
    {"name": "EcoWear", "materials": ["organic cotton", "recycled polyester"]},
    {"name": "GreenThreads", "materials": ["organic cotton", "bamboo"]},
    {"name": "SustainableStyle", "materials": ["bamboo", "recycled polyester"]}
]

brands_2 = [
    {"name": "NatureWear", "materials": ["hemp", "linen"]},
    {"name": "Earthly", "materials": ["organic cotton", "hemp"]},
    {"name": "GreenFit", "materials": ["linen", "recycled wool"]}
]

brands_3 = [
    {"name": "OrganicThreads", "materials": ["organic cotton"]},
    {"name": "EcoFashion", "materials": ["recycled polyester", "hemp"]},
    {"name": "GreenLife", "materials": ["recycled polyester", "bamboo"]}
]

print(find_trending_materials(brands))
print(find_trending_materials(brands_2))
print(find_trending_materials(brands_3))
```

Example Output:

```
['organic cotton', 'recycled polyester', 'bamboo']
['hemp', 'linen']
['recycled polyester']
```

## Problem 4: Fabric Pairing

You want to find pairs of fabrics that, when combined, maximize eco-friendliness while staying within a budget. Each fabric has a cost associated with it, and your goal is to identify the pair of fabrics whose combined cost is the highest possible without exceeding the budget.

Write the `find_best_fabric_pair()` function, which takes a list of fabrics (each with a name and cost) and a budget. The function should return the names of the two fabrics whose combined cost is the closest to the budget without exceeding it.

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_best_fabric_pair(fabrics, budget):
    pass
```

Example Usage:

```
fabrics = [("Organic Cotton", 30), ("Recycled Polyester", 20), ("Bamboo", 25), ("Hemp", 15)]
fabrics_2 = [("Linen", 50), ("Recycled Wool", 40), ("Tencel", 30), ("Organic Cotton", 60)]
fabrics_3 = [("Linen", 40), ("Hemp", 35), ("Recycled Polyester", 25), ("Bamboo", 20)]

print(find_best_fabric_pair(fabrics, 45))
print(find_best_fabric_pair(fabrics_2, 70))
print(find_best_fabric_pair(fabrics_3, 60))
```

Example Output:

```
('Hemp', 'Organic Cotton')
('Tencel', 'Recycled Wool')
('Bamboo', 'Linen')
```

## Problem 5: Fabric Stacks

You need to organize rolls of fabric in such a way that you can efficiently retrieve them based on their eco-friendliness rating. Fabrics are stacked one on top of the other, and you can only retrieve the top fabric in the stack.

Write the `organize_fabrics()` function, which takes a list of fabrics (each with a name and an eco-friendliness rating) and returns a list of fabric names in the order they would be retrieved from the stack, starting with the least eco-friendly fabric.

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 organize_fabrics(fabrics):
    pass
```

Example Usage:

```
fabrics = [("Organic Cotton", 8), ("Recycled Polyester", 6), ("Bamboo", 7), ("Hemp", 9)]
fabrics_2 = [("Linen", 5), ("Recycled Wool", 9), ("Tencel", 7), ("Organic Cotton", 6)]
fabrics_3 = [("Linen", 4), ("Hemp", 8), ("Recycled Polyester", 5), ("Bamboo", 7)]

print(organize_fabrics(fabrics))
print(organize_fabrics(fabrics_2))
print(organize_fabrics(fabrics_3))
```

Example Output:

```
['Hemp', 'Organic Cotton', 'Bamboo', 'Recycled Polyester']
['Recycled Wool', 'Tencel', 'Organic Cotton', 'Linen']
['Hemp', 'Bamboo', 'Recycled Polyester', 'Linen']
```

## Problem 6: Supply Chain

In the sustainable fashion industry, managing the supply chain efficiently is crucial. Supplies arrive in a sequence, and you need to process them in the order they arrive. However, some supplies may be of higher priority due to their eco-friendliness or scarcity.

Write the `process_supplies()` function, which takes a list of supplies (each with a name and a priority level) and returns a list of supply names in the order they would be processed, with higher priority supplies processed first.

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 process_supplies(supplies):
    pass
```

Example Usage:

```
supplies = [("Organic Cotton", 3), ("Recycled Polyester", 2), ("Bamboo", 4), ("Hemp", 1)]
supplies_2 = [("Linen", 2), ("Recycled Wool", 5), ("Tencel", 3), ("Organic Cotton", 4)]
supplies_3 = [("Linen", 3), ("Hemp", 2), ("Recycled Polyester", 5), ("Bamboo", 1)]

print(process_supplies(supplies))
print(process_supplies(supplies_2))
print(process_supplies(supplies_3))
```

Example Output:

```
['Bamboo', 'Organic Cotton', 'Recycled Polyester', 'Hemp']
['Recycled Wool', 'Organic Cotton', 'Tencel', 'Linen']
['Recycled Polyester', 'Linen', 'Hemp', 'Bamboo']
```

# Problem 7: Calculate Fabric Waste

In the sustainable fashion industry, minimizing waste is crucial. After cutting out patterns for clothing items, there are often leftover pieces of fabric that cannot be used. Your task is to calculate the total amount of fabric waste generated after producing a collection of clothing items. Each clothing item requires a certain amount of fabric, and the available fabric rolls come in fixed lengths.

Write the `calculate_fabric_waste()` function, which takes a list of clothing items (each with a required fabric length) and a list of fabric rolls (each with a specific length). The function should return the total fabric waste after producing all the items.

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 calculate_fabric_waste(items, fabric_rolls):
    pass
```

Example Usage:

```
items = [("T-Shirt", 2), ("Pants", 3), ("Jacket", 5)]
fabric_rolls = [5, 5, 5]

items_2 = [("Dress", 4), ("Skirt", 3), ("Blouse", 2)]
fabric_rolls = [4, 4, 4]

items_3 = [("Jacket", 6), ("Shirt", 2), ("Shorts", 3)]
fabric_rolls = [7, 5, 5]

print(calculate_fabric_waste(items, fabric_rolls))
print(calculate_fabric_waste(items_2, fabric_rolls))
print(calculate_fabric_waste(items_3, fabric_rolls))
```

Example Output:

```
5
3
6
```

# Problem 8: Fabric Roll Organizer

You need to organize fabric rolls for optimal usage. Each fabric roll has a specific length, and you want to group them into pairs so that the difference between the lengths of the rolls in each pair is minimized. If there's an odd number of rolls, one roll will be left out.

Write the `organize_fabric_rolls()` function, which takes a list of fabric roll lengths and returns a pair of fabric roll lengths, where the difference in lengths between the rolls is minimized. If there's an odd number of rolls, the last roll should be returned separately.

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 organize_fabric_rolls(fabric_rolls):
    pass
```

Example Usage:

```
fabric_rolls = [15, 10, 25, 30, 22]
fabric_rolls_2 = [5, 8, 10, 7, 12, 14]
fabric_rolls_3 = [40, 10, 25, 15, 30]

print(organize_fabric_rolls(fabric_rolls))
print(organize_fabric_rolls(fabric_rolls_2))
print(organize_fabric_rolls(fabric_rolls_3))
```

Example Output:

```
[(10, 15), (22, 25), 30]
[(5, 7), (8, 10), (12, 14)]
[(10, 15), (25, 30), 40]
```

Close Section

## ▾ Advanced Problem Set Version 2

### Problem 1: Track Screen Time Usage

In the digital age, managing screen time is crucial for maintaining a healthy balance between online and offline activities. You need to track how much time users spend on different apps throughout the day.

Write the `track_screen_time()` function, which takes a list of logs, where each log contains an app name and the number of minutes spent on that app during a specific hour. The function should return the app names and the total time spent on each app.

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 track_screen_time(logs):
    pass
```

Example Usage:

```
logs = [("Instagram", 30), ("YouTube", 20), ("Instagram", 25), ("Snapchat", 15), ("YouTube",
logs_2 = [("Twitter", 10), ("Reddit", 20), ("Twitter", 15), ("Instagram", 35)]
logs_3 = [("TikTok", 40), ("TikTok", 50), ("YouTube", 60), ("Snapchat", 25)]

print(track_screen_time(logs))
print(track_screen_time(logs_2))
print(track_screen_time(logs_3))
```

Example Output:

```
{'Instagram': 55, 'YouTube': 30, 'Snapchat': 15}
{'Twitter': 25, 'Reddit': 20, 'Instagram': 35}
{'TikTok': 90, 'YouTube': 60, 'Snapchat': 25}
```

### ▼ 💡 Hint: Big O (Time & Space Complexity)

Big O notation is a mathematical notation in computer science used to describe the the time and space complexity of an algorithm. Time complexity is the amount of time an algorithm or function takes to run in comparison to the size of the input data. Space complexity is the amount of extra memory or space an algorithm or function needs to complete its task in comparison to the size of the input data.

For your convenience, we've included a summary of the three most common Big O functions below.

Common Big O includes:

- **O(1) - Constant Time** No matter the size of your input data, the function takes a fixed amount of time or memory to complete its task.

  Example: Summing two numbers

  ```
  def sum(a, b):
      return a + b
  ```

  It takes the computer roughly the same amount of time to sum `a` and `b` no matter how large the two numbers are.

- **O(n) - Linear Time** The amount of time or memory your function needs grows linearly with the size of your input data.

  Example: Printing each item in a list

  ```
  def print_list(lst):
      for item in lst:
          print(item)
  ```

  The computer has to perform one extra print statement for each extra item there is in the list, so the length of time it takes to print the list will be proportional to the number of items in the list. We expect that it will take 1000 times longer to print a list with 1000

elements than it will to print a list with just 1 element.

- **O(n²) - Quadratic Time** The amount of time or memory your function needs grows quadratically with the size of your input data.

  Example: Finding Duplicates Using a Nested For Loop

  ```
  def find_duplicates(lst):
      n = len(lst)
      for i in range(n):
          for j in range(i + 1, n):
              if lst[i] == lst[j]:
                  print(f"Duplicate found: {lst[i]}")
                  return True
      return False
  ```

  The function compares each element in the list to every other element in the list, which means we perform roughly n² comparisons where n is the length of our input list `lst`, so it will take n² time to complete all comparisons. We can expect that for a list of size 2, we will perform roughly 4 comparisons whereas for a list of size 10 we will perform roughly 100 comparisons.

### ▼ ✨ AI Hint: Decoding Big O

*Key Skill: Use AI to explain code concepts*

Big O is a big topic, and kind of tricky to wrap your head around! If you're feeling confused, try asking an AI tool like ChatGPT or GitHub Copilot to explain it to you:

*"You're an expert computer science tutor for a Python-based technical interviewing prep course. Can you use an analogy to help me understand Big O notation? Please explain the concept of time and space complexity in a way that is easy to understand."*

Once it gives you an answer, you can ask follow-up questions to clarify any points that are still confusing. Be patient with yourself, and remember that this is a complex topic that takes time to fully understand!

## Problem 2: Identify Most Used Apps

You want to help users identify which apps they spend the most time on throughout the day. Based on the screen time logs, your task is to find the app with the highest total screen time.

Write the `most_used_app()` function, which takes a dictionary containing the app names and the total time spent on each app. The function should return the app with the highest screen time. If multiple apps have the same highest screen time, return any one of them.

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 most_used_app(screen_time):
    pass
```

Example Usage:

```
screen_time = {"Instagram": 55, "YouTube": 30, "Snapchat": 15}
screen_time_2 = {"Twitter": 25, "Reddit": 20, "Instagram": 35}
screen_time_3 = {"TikTok": 90, "YouTube": 90, "Snapchat": 25}

print(most_used_app(screen_time))
print(most_used_app(screen_time_2))
print(most_used_app(screen_time_3))
```

Example Output:

```
Instagram
Instagram
TikTok
```

## Problem 3: Weekly App Usage

Users want to know how much time they are spending on each app over the course of a week. Your task is to summarize the total weekly usage for each app and then identify the app with the most varied usage pattern throughout the week. The varied usage pattern can be measured by the difference between the maximum and minimum daily usage for each app.

Write the `most_varied_app()` function, which takes a dictionary containing the app names and daily usage over seven days. The function should return the app with the highest difference between the maximum and minimum usage over the week. If multiple apps have the same difference, return any one of them.

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 most_varied_app(app_usage):
    pass
```

Example Usage:

```
app_usage = {
    "Instagram": [60, 55, 65, 60, 70, 55, 60],
    "YouTube": [100, 120, 110, 100, 115, 105, 120],
    "Snapchat": [30, 35, 25, 30, 40, 35, 30]
}

app_usage_2 = {
    "Twitter": [15, 15, 15, 15, 15, 15, 15],
    "Reddit": [45, 50, 55, 50, 45, 50, 55],
    "Facebook": [80, 85, 80, 85, 80, 85, 80]
}

app_usage_3 = {
    "TikTok": [80, 100, 90, 85, 95, 105, 90],
    "Spotify": [40, 45, 50, 45, 40, 45, 50],
    "WhatsApp": [60, 60, 60, 60, 60, 60, 60]
}

print(most_varied_app(app_usage))
print(most_varied_app(app_usage_2))
print(most_varied_app(app_usage_3))
```

Example Output:

```
YouTube
Reddit
TikTok
```

## Problem 4: Daily App Usage Peaks

You want to help users identify the peak hours of their app usage during the day. Users log their app usage every hour, and your task is to determine the highest total screen time recorded during any three consecutive hours.

Write the `peak_usage_hours()` function that takes a list of 24 integers, where each integer represents the number of minutes spent on apps during a specific hour (from hour 0 to hour 23). The function should return the start hour and the total screen time for the three-hour period with the highest total usage. If multiple periods have the same total, return the earliest one.

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 peak_usage_hours(screen_time):
  pass
```

Example Usage:

```
screen_time = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 1
screen_time_2 = [5, 15, 10, 20, 30, 25, 50, 40, 35, 45, 60, 55, 65, 75, 70, 85, 95, 90, 100,
screen_time_3 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

print(peak_usage_hours(screen_time))
print(peak_usage_hours(screen_time_2))
print(peak_usage_hours(screen_time_3))
```

Example Output:

```
(21, 690)
(21, 360)
(0, 0)
```

## Problem 5: App Usage Pattern Recognition

Users want to identify patterns in their app usage over the course of a day. Specifically, they are interested in finding out if they have periods of repetitive behavior, where they switch between the same set of apps in a recurring pattern. Your task is to detect the longest repeating pattern of app usage within a 24-hour period.

Write the `find_longest_repeating_pattern()` function, which takes a list of app usage logs, where each element in the list represents the app used in a particular hour (from hour 0 to hour 23). The function should return the longest repeating pattern of apps and how many times the pattern repeats. If there are multiple patterns of the same length, return the first one found.

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_longest_repeating_pattern(app_logs):
  pass
```

Example Usage:

```
app_logs = ["Instagram", "YouTube", "Snapchat", "Instagram", "YouTube", "Snapchat", "Instagr
app_logs_2 = ["Facebook", "Instagram", "Facebook", "Instagram", "Facebook", "Instagram", "Sn
app_logs_3 = ["WhatsApp", "TikTok", "Instagram", "YouTube", "Snapchat", "Twitter", "Facebook

print(find_longest_repeating_pattern(app_logs))
print(find_longest_repeating_pattern(app_logs_2))
print(find_longest_repeating_pattern(app_logs_3))
```

Example Output:

```
(['Instagram', 'YouTube', 'Snapchat'], 3)
(['Facebook', 'Instagram'], 3)
(['WhatsApp', 'TikTok', 'Instagram', 'YouTube', 'Snapchat', 'Twitter', 'Facebook'], 2)
```

# Problem 6: Screen Time Session Management

As part of a digital wellbeing initiative, you're designing a system to manage screen time sessions on a device. The device tracks various apps being opened and closed throughout the day. Each app opening starts a new session, and each closing ends that session. The system should ensure that for every app opened, there is a corresponding closure.

Write the `manage_screen_time_sessions()` function, which takes a list of actions representing app openings and closures throughout the day. Each action is either `"OPEN <app>"` or `"CLOSE <app>"`. The function should return `True` if all the app sessions are properly managed (i.e., every opened app has a corresponding close in the correct order), and `False` otherwise.

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 manage_screen_time_sessions(actions):
  pass
```

Example Usage:

```
actions = ["OPEN Instagram", "OPEN Facebook", "CLOSE Facebook", "CLOSE Instagram"]
actions_2 = ["OPEN Instagram", "CLOSE Instagram", "CLOSE Facebook"]
actions_3 = ["OPEN Instagram", "OPEN Facebook", "CLOSE Instagram", "CLOSE Facebook"]

print(manage_screen_time_sessions(actions))
print(manage_screen_time_sessions(actions_2))
print(manage_screen_time_sessions(actions_3))
```

Example Output:

```
True
False
False
```

# Problem 7: Digital Wellbeing Dashboard Analysis

You're building a digital wellbeing dashboard that tracks users' daily app usage and helps them identify patterns and areas for improvement. Each user has a log of their daily app usage, which includes various activities like Social Media, Entertainment, Productivity, and so on. The goal is to analyze this data to provide insights into their usage patterns.

Write the `analyze_weekly_usage()` function, which takes a dictionary where each key is a day of the week (e.g., `"Monday"`, `"Tuesday"`) and the value is another dictionary. This nested dictionary's keys represent app categories (e.g., `"Social Media"`, `"Entertainment"`) and its values represent the time spent (in minutes) on that category during that day.

Your function should return:

1. The total time spent on each category across the entire week.
2. The day with the highest total usage.
3. The most-used category of the week.

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 analyze_weekly_usage(weekly_usage):
  pass
```

Example Usage:

```
weekly_usage = {
    "Monday": {"Social Media": 120, "Entertainment": 60, "Productivity": 90},
    "Tuesday": {"Social Media": 100, "Entertainment": 80, "Productivity": 70},
    "Wednesday": {"Social Media": 130, "Entertainment": 70, "Productivity": 60},
    "Thursday": {"Social Media": 90, "Entertainment": 60, "Productivity": 80},
    "Friday": {"Social Media": 110, "Entertainment": 100, "Productivity": 50},
    "Saturday": {"Social Media": 180, "Entertainment": 120, "Productivity": 40},
    "Sunday": {"Social Media": 160, "Entertainment": 140, "Productivity": 30}
}
print(analyze_weekly_usage(weekly_usage))
```

Example Output:

```
{'total_category_usage': {'Social Media': 890, 'Entertainment': 630, 'Productivity': 420}, 'h
```

## Problem 8: Optimizing Break Times

As part of a digital wellbeing initiative, your goal is to help users optimize their break times throughout the day. Users have a list of activities they perform during breaks, each with a specified duration in minutes. You want to find two breaks that have the total duration closest to a target time.

Write the `find_best_break_pair()` function, which takes a list of integers representing the duration of each break in minutes and a target time in minutes. The function should return the pair of break durations that sum closest to the target time. If there are multiple pairs with the same closest sum, return the pair with the smallest break durations. If the list has fewer than two breaks, return an empty tuple.

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_best_break_pair(break_times, target):
  pass
```

Example Usage:

```
break_times = [10, 20, 35, 40, 50]
break_times_2 = [5, 10, 25, 30, 45]
break_times_3 = [15, 25, 35, 45]
break_times_4 = [30]

print(find_best_break_pair(break_times, 60))
print(find_best_break_pair(break_times_2, 50))
print(find_best_break_pair(break_times_3, 70))
print(find_best_break_pair(break_times_4, 60))
```

Example Output:

```
(20, 40)
(5, 45)
(25, 45)
()
```

Close Section