TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Session 2: Dictionaries & Sets

Session Overview

Students will continue to expand their expertise in Python through the exploration of data structures such as lists, dictionaries, and sets. They engage in various tasks like verifying list properties, creating and updating dictionaries, and analyzing data to make decisions.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

🎢 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
    - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

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

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 2

▶ Standard Problem Set Version 1

▶ Standard Problem Set Version 2

▼ Advanced Problem Set Version 1

Problem 1: Balanced Art Collection

As the curator of an art gallery, you are organizing a new exhibition. You must ensure the collection of art pieces are balanced to attract the right range of buyers. A balanced collection is one where the difference between the maximum and minimum value of the art pieces is exactly 1.

Given an integer array `art_pieces` representing the value of each art piece, write a function `find_balanced_subsequence()` that returns the length of the longest balanced subsequence.

A **subsequence** is a sequence derived from the array by deleting some or no elements without changing the order of the remaining elements.

```
def find_balanced_subsequence(art_pieces):
    pass
```

Example Usage:

```
art_pieces1 = [1,3,2,2,5,2,3,7]
art_pieces2 = [1,2,3,4]
art_pieces3 = [1,1,1,1]

print(find_balanced_subsequence(art_pieces1))
print(find_balanced_subsequence(art_pieces2))
print(find_balanced_subsequence(art_pieces3))
```

Example Output:

```
5
Example 1 Explanation:  The longest balanced subsequence is [3,2,2,2,3].

2
0
```

Problem 2: Verifying Authenticity

Your art gallery has just been shipped a new collection of numbered art pieces, and you need to verify their authenticity. The collection is considered "authentic" if it is a permutation of an array `base[n]`.

The `base[n]` array is defined as `[1, 2, ..., n - 1, n, n]`, meaning it is an array of length `n + 1` containing the integers from `1` to `n - 1` exactly once, and the integer `n` twice. For example, `base[1]` is `[1, 1]` and `base[3]` is `[1, 2, 3, 3]`.

Write a function `is_authentic_collection` that accepts an array of integers `art_pieces` and returns `True` if the given array is an authentic array, and otherwise returns `False`.

Note: A permutation of integers represents an arrangement of these numbers. For example `[3, 2, 1]` and `[2, 1, 3]` are both permutations of the series of numbers `1`, `2`, and `3`.

```
def is_authentic_collection(art_pieces):
    pass
```

Example Usage:

```
collection1 = [2, 1, 3]
collection2 = [1, 3, 3, 2]
collection3 = [1, 1]

print(is_authentic_collection(collection1))
print(is_authentic_collection(collection2))
print(is_authentic_collection(collection3))
```

Example Output:

```
False
Example 1 Explanation: Since the maximum element of the array is 3, the only
candidate n for which this array could be a permutation of base[n], is n = 3.
However, base[3] has four elements but array collection1 has three. Therefore,
it can not be a permutation of base[3] = [1, 2, 3, 3]. So the answer is false.

True
Example 2 Explanation:  Since the maximum element of the array is 3, the only
candidate n for which this array could be a permutation of base[n], is n = 3.
It can be seen that collection2 is a permutation of base[3] = [1, 2, 3, 3]
(by swapping the second and fourth elements in nums, we reach base[3]).
 Therefore, the answer is true.

True
Example 3 Explanation; Since the maximum element of the array is 1,
the only candidate n for which this array could be a permutation of base[n],
is n = 1. It can be seen that collection3 is a permutation of base[1] = [1, 1].
 Therefore, the answer is true.
```

Problem 3: Gallery Wall

You are tasked with organizing a collection of art prints represented by a list of strings `collection`. You need to display these prints on a single wall in a 2D array format that meets the following criteria:

1. The 2D array should contain only the elements of the array `collection`.
2. Each row in the 2D array should contain distinct strings.
3. The number of rows in the 2D array should be minimal.

Return the resulting array. If there are multiple answers, return any of them. Note that the 2D array can have a different number of elements on each row.

```
def organize_exhibition(collection):
    pass
```

Example Usage:

```
collection1 = ["O'Keefe", "Kahlo", "Picasso", "O'Keefe", "Warhol",
               "Kahlo", "O'Keefe"]
collection2 = ["Kusama", "Monet", "Ofili", "Banksy"]

print(organize_exhibition(collection1))
print(organize_exhibition(collection2))
```

Example Output:

```
[
  ["O'Keefe", "Kahlo", "Picasso", "Warhol"],
  ["O'Keefe", "Kahlo"],
  ["O'Keefe"]
]
Example 1 Explanation:
All elements of collections were used, and each row of the 2D array contains
distinct strings, so it is a valid answer.
It can be shown that we cannot have less than 3 rows in a valid array.

[["Kusama", "Monet", "Ofili", "Banksy"]]
Example 2 Explanation:
All elements of the array are distinct, so we can keep all of them in the first
row of the 2D array.
```

Problem 4: Gallery Subdomain Traffic

Your gallery has been trying to increase it's online presence by hosting several virtual galleries. Each virtual gallery's web traffic is tracked through domain names, where each domain may have subdomains.

A domain like `"modern.artmuseum.com"` consists of various subdomains. At the top level, we have `"com"`, at the next level, we have `"artmuseum.com"`, and at the lowest level, `"modern.artmuseum.com"`. When visitors access a domain like `"modern.artmuseum.com"`, they also implicitly visit the parent domains `"artmuseum.com"` and `"com"`.

A **count-paired domain** is represented as `"rep d1.d2.d3"` where `rep` is the number of visits to the domain and `d1.d2.d3` is the domain itself.

- For example, `"9001 modern.artmuseum.com"` indicates that `"modern.artmuseum.com"` was visited `9001` times.

Given an array of count-paired domains `cpdomains`, return an array of the count-paired domains of each subdomain. The order of the output does not matter.

```
def subdomain_visits(cpdomains):
    pass
```

Example Usage:

```
cpdomains1 = ["9001 modern.artmuseum.com"]
cpdomains2 = ["900 abstract.gallery.com", "50 impressionism.com",
              "1 contemporary.gallery.com", "5 medieval.org"]

print(subdomain_visits(cpdomains1))
print(subdomain_visits(cpdomains2))
```

Example Output:

```
["9001 artmuseum.com", "9001 modern.artmuseum.com", "9001 com"]

["901 gallery.com", "50 impressionism.com", "900 abstract.gallery.com", "5 medieval.
"1 contemporary.gallery.com", "951 com"]
```

Problem 5: Beautiful Collection

Your gallery has entered a competition for the most beautiful collection. Your collection is represented by a string `collection` where each artist in your gallery is represented by a character. The beauty of a collection is defined as the difference in frequencies between the most frequent and least frequent characters.

- For example, the beauty of `"abaacc"` is `3 - 1 = 2`.

Given a string `collection`, write a function `beauty_sum()` that returns *the sum of beauty of all of its substrings (subcollections)*, not just of the collection itself.

```
def beauty_sum(collection):
    pass
```

Example Usage:

```
print(beauty_sum("aabcb"))
print(beauty_sum("aabcbaa"))
```

Example Output:

```
5
Example 1 Explanation: The substrings with non-zero beauty are
["aab","aabc","aabcb","abcb","bcb"], each with beauty equal to 1.

17
```

Problem 6: Counting Divisible Collections in the Gallery

You have a list of integers `collection_sizes` representing the sizes of different art collections in your gallery and are trying to determine how to group them to best fit in your space. Given an integer `k` write a function `count_divisible_collections()` that returns the number of non-empty subarrays (contiguous parts of the array) where the sum of the sizes is divisible by `k`.

```
def count_divisible_collections(collection_sizes, k):
    pass
```

Example Usage:

```
nums1 = [4, 5, 0, -2, -3, 1]
k1 = 5
nums2 = [5]
k2 = 9

print(count_divisible_collections(nums1, k1))
print(count_divisible_collections(nums2, k2))
```

Example Output:

```
7
Example 1 Explanation: There are 7 subarrays with a sum divisible by k = 5:
[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

0
```

Close Section

▼ Advanced Problem Set Version 2

Problem 1: Cook Off

In a reality TV show, contestants are challenged to do the best recreation of a meal cooked by an all-star judge using limited resources. The meal they must recreate is represented by the string `target_meal`. The contestants are given a collection of ingredients represented by the string `ingredients`.

Help the contestants by writing a function `max_attempts()` that returns the maximum number of copies of `target_meal` they can create using the given `ingredients`. You can take some letters from `ingredients` and rearrange them to form new strings.

```
def max_attempts(ingredients, target_meal):
    pass
```

Example Input:

```
ingredients1 = "aabbbcccc"
target_meal1 = "abc"

ingredients2 = "ppppqqqrrr"
target_meal2 = "pqr"

ingredients3 = "ingredientsforcooking"
target_meal3 = "cooking"
```

Example Output:

```
2
3
1
```

▶ 💡 **Hint: Representing Infinite Values**

Problem 2: Dialogue Similarity

Watching a reality TV show, you notice a lot of contestants talk similarly. We want to determine if two contestants have similar speech patterns.

We can represent a sentence as an array of words, for example, the sentence `"I've got a text!"` can be represented as `sentence = ["I've", "got", "a", "text"]`.

You are given two sentences from different contestants `sentence1` and `sentence2` each represented as a string array and given an array of string pairs `similar_pairs` where `similar_pairs[i] = [xi, yi]` indicates that the two words `xi` and `yi` are similar. Write a function `is_similar()` that returns `True` if `sentence1` and `sentence2` are similar, and `False` if they are not similar.

Two sentences are similar if:

- They have **the same length** (i.e., the same number of words)
- `sentence1[i]` and `sentence2[i]` are similar

Notice that a word is always similar to itself, also notice that the similarity relation is not transitive. For example, if the words `a` and `b` are similar, and the words `b` and `c` are similar, `a` and `c` are not necessarily similar.

```
def is_similar(sentence1, sentence2, similar_pairs):
    pass
```

Example Usage:

```
sentence1 = ["my", "type", "on", "paper"]
sentence2 = ["my", "type", "in", "theory"]
similar_pairs = [ ["on", "in"], ["paper", "theory"]]

sentence3 = ["no", "tea", "no", "shade"]
sentence4 = ["no", "offense"]
similar_pairs2 = [["shade", "offense"]]

print(is_similar(sentence1, sentence2, similar_pairs))
print(is_similar(sentence3, sentence4, similar_pairs2))
```

Example Output:

```
True
Example 1 Explanation: "my" and "type" are similar to themselves. The words at
indices 2 and 3 of sentence1 are similar to words at indices 2 and 3 of
sentence2 according to the similar_pairs array.

False
Example 2 Explanation: Sentences are of different length.
```

Problem 3: Cows and Bulls

In a reality TV show, contestants play a mini-game called Bulls and Cows for a prize. The objective is to guess a secret number within a limited number of attempts. You, as the host, need to provide hints to the contestants based on their guesses.

When a contestant makes a guess, you provide a hint with the following information:

- The number of "bulls," which are digits in the guess that are in the correct position.
- The number of "cows," which are digits in the guess that are in the secret number but are located in the wrong position.

Given the secret number `secret` and the contestant's guess `guess`, return the hint for their guess.

The hint should be formatted as `"xAyB"`, where `x` is the number of bulls and `y` is the number of cows. Note that both `secret` and `guess` may contain duplicate digits.

```
def get_hint(secret, guess):
    pass
```

Example Input:

```
secret1 = "1807"
guess1 = "7810"

secret2 = "1123"
guess2 = "0111"

print(get_hint(secret1, guess1))
print(get_hint(secret2, guess2))
```

Example Output:

```
1A3B
Example 1 Explanation:
Bulls are connected with a '|' and cows are marked with an asterisk:
"1807"
  |
"7810"
 * **

1A1B
Example 2 Explanation:
Bulls are connected with a '|' and cows are marked with an asterisk:
"1123"        "1123"
  |      or     |
"0111"        "0111"
   *             *
Note that only one of the two unmatched 1s is counted as a cow since the
non-bull digits can only be rearranged to allow one 1 to be a bull.
```

Problem 4: Count Winning Pairings

In a popular reality TV show, contestants pair up for various challenges. The pairing is considered winning if the sum of their "star power" is a power of two.

You are given an array of integers `star_power` where `star_power[i]` represents the star power of the i-th contestant. Return the number of different winning pairings you can make from this list, modulo `10^9 + 7`.

Note that contestants with different indices are considered different even if they have the same star power.

```python
def count_winning_pairings(star_power):
    pass
```

Example Usage:

```python
star_power1 = [1, 3, 5, 7, 9]
print(count_winning_pairings(star_power1))

star_power2 = [1, 1, 1, 3, 3, 3, 7]
print(count_winning_pairings(star_power2))
```

Example Output:

```
4
15
```

Problem 5: Assigning Unique Nicknames to Contestants

In a reality TV show, contestants are assigned unique nicknames. However, two contestants cannot have the same nickname. If a contestant requests a nickname that has already been taken, the show will add a suffix to the name in the form of `(k)`, where `k` is the smallest positive integer that makes the nickname unique.

You are given an array of strings `nicknames` representing the requested nicknames for the contestants. Return an array of strings where `result[i]` is the actual nickname assigned to the `i` th contestant.

```
def assign_unique_nicknames(nicknames):
    pass
```

Example Usage:

```
nicknames1 = ["Champ","Diva","Champ","Ace"]
print(assign_unique_nicknames(nicknames1))

nicknames2 = ["Ace","Ace","Ace","Maverick"]
print(assign_unique_nicknames(nicknames2))

nicknames3 = ["Star","Star","Star","Star","Star"]
print(assign_unique_nicknames(nicknames3))
```

Example Output:

```
["Champ","Diva","Champ(1)","Ace"]
["Ace","Ace(1)","Ace(2)","Maverick"]
["Star","Star(1)","Star(2)","Star(3)","Star(4)"]
```

Problem 6: Pair Contestants

In a reality TV challenge, contestants must be paired up in teams. Each team's combined score must be divisible by a target number `k`. You are given an array of integers `scores` representing the scores of the contestants and an integer `k`.

You need to determine whether it is possible to pair all contestants such that the sum of the scores of each pair is divisible by `k`.

Return `True` if it is possible to form the required pairs, otherwise return `False`.

```
def pair_contestants(scores, k):
    pass
```

Example Usage:

```
scores1 = [1,2,3,4,5,10,6,7,8,9]
k1 = 5
print(pair_contestants(scores1, k1))

scores2 = [1,2,3,4,5,6]
k2 = 7
print(pair_contestants(scores2, k2))

scores3 = [1,2,3,4,5,6]
k3 = 10
print(pair_contestants(scores3, k3))
```

Example Output:

```
True
True
False
```

Close Section