TIP102 | Intermediate Technical Interview Prep

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

Session 2: 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

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 2

▾ Standard Problem Set Version 1

Problem 1: Most Endangered Species

Imagine you are working on a wildlife conservation database. Write a function named `most_endangered()` that returns the species with the highest conservation priority based on its population.

The function should take in a list of dictionaries named `species_list` as a parameter. Each dictionary represents data associated with a species, including its `name`, `habitat`, and wild `population`. The function should return the `name` of the species with the lowest `population`.

If there are multiple species with the lowest population, return the species with the *lowest* index.

```
def most_endangered(species_list):
    pass
```

Example Usage:

```
species_list = [
    {"name": "Amur Leopard",
     "habitat": "Temperate forests",
     "population": 84
    },
    {"name": "Javan Rhino",
     "habitat": "Tropical forests",
     "population": 72
    },
    {"name": "Vaquita",
     "habitat": "Marine",
     "population": 10
    }
]

print(most_endangered(species_list))
```

Example Output:

```
Vaquita
```

▼ ✨ AI Hint: Accessing Values in a Dictionary

Key Skill: Use AI to explain code concepts

This question will require you to use keys to access their corresponding values in a dictionary. There are two common ways to access values in a dictionary. Try asking ChatGPT or GitHub copilot:

"You're an expert computer science tutor. Please show me the two most common ways to access values in a dictionary in Python, and explain how each one works."

Then open the next hint to see the answer!

▼ 💡 Hint: Dictionary Access options

The two common ways to access values in a dictionary are square bracket notation `d[key]` and the `get()` method.

The Unit 2 cheatsheet includes a more thorough breakdown of these two options. If you still feel confused after reviewing the cheatsheet, try asking generative AI to help you understand!

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

This question will require you to loop over a dictionary. We have three options for looping over a dictionary: looping over the keys, values, or key-value pairs. To explore how to access the keys, values, and key-value pairs reference the unit cheatsheet. For specific examples of looping over a dictionary, ask a generative AI tool to provide an example or search for existing examples using a search engine.

Problem 2: Identifying Endangered Species

As part of conservation efforts, certain species are considered endangered and are represented by the string `endangered_species`. Each character in this string denotes a different endangered species. You also have a record of all observed species in a particular region, represented by the string `observed_species`. Each character in `observed_species` denotes a species observed in the region.

Your task is to determine how many instances of the observed species are also considered endangered.

Note: Species are case-sensitive, so "a" is considered a different species from "A".

Write a function to count the number of endangered species observed.

```
def count_endangered_species(endangered_species, observed_species):
    pass
```

Example Usage:

```
endangered_species1 = "aA"
observed_species1 = "aAAbbbb"

endangered_species2 = "z"
observed_species2 = "ZZ"

print(count_endangered_species(endangered_species1, observed_species1))
print(count_endangered_species(endangered_species2, observed_species2))
```

Example Output:

```
3 # `a` and `A` are endangered species. `a` appears once, and `A` twice.
0
```

▼ ✨ **AI Hint: Introduction to sets**

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **set**. A Python set is a data type which holds an unordered, mutable collection of *unique* elements.

If you are unfamiliar with what a set is, or how to create a set, you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a set is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a set different from a list or dictionary? Can you show me examples of each?"

Problem 3: Navigating the Research Station

In a wildlife research station, each letter of the alphabet represents a different observation point laid out in a single row. Given a string `station_layout` of length `26` indicating the layout of these observation points (indexed from `0` to `25`), you start your journey at the first observation point (index `0`). To make observations in a specific order represented by a string `observations`, you need to move from one point to another.

The time taken to move from one observation point to another is the absolute difference between their indices, `|i - j|`.

Write a function that returns the total time it takes to visit all the required observation points in the given order with one movement.

```
def navigate_research_station(station_layout, observations):
  pass
```

Example Usage:

```
station_layout1 = "pqrstuvwxyzabcdefghijklmno"
observations1 = "wildlife"

station_layout2 = "abcdefghijklmnopqrstuvwxyz"
observations2 = "cba"

print(navigate_research_station(station_layout1, observations1))
print(navigate_research_station(station_layout2, observations2))
```

Example Output:

```
45
4
Example 2 explanation: The index moves from 0 to 2 to observe 'c', then to 1 for
'b', then to 0 again for 'a'.
Total time = 2 + 1 + 1 = 4.
```

▼ 💡 **Hint: What should my keys and values be?**

When considering whether we can solve a problem with dictionaries, we want to consider what our keys and corresponding values could possibly be. In this case, we could match characters to their indices in `station_layout`.

▼ ✨ **AI Hint: `enumerate()`**

Key Skill: Use AI to explain code concepts

This problem may benefit from use of the `enumerate()` function. For a quick refresher on how the `enumerate()` function works, check out the Unit 2 Cheatsheet.

If you'd still like to see more examples or ask follow-up questions, try using an AI tool like ChatGPT or GitHub Copilot. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Please provide 2-3 examples of how the `enumerate()` function is used in Python, and explain how each one works."

Problem 4: Prioritizing Endangered Species Observations

In your work with a wildlife conservation database, you have two lists: `observed_species` and `priority_species`. The elements of `priority_species` are distinct, and all elements in `priority_species` are also in `observed_species`.

Write a function `prioritize_observations()` that sorts the elements of `observed_species` such that the relative ordering of items in `observed_species` matches that of `priority_species`. Species that do not appear in `priority_species` should be placed at the end of `observed_species` in ascending order.

```
def prioritize_observations(observed_species, priority_species):
  pass
```

Example Usage:

```python
observed_species1 = ["🐯", "🦁", "🦌", "🦁", "🐯", "🐘", "🐍", "🦑", "🐻", "🐯", "🐼"]
priority_species1 = ["🐯", "🦌", "🐘", "🦁"]

observed_species2 = ["bluejay", "sparrow", "cardinal", "robin", "crow"]
priority_species2 = ["cardinal", "sparrow", "bluejay"]

print(prioritize_observations(observed_species1, priority_species1))
print(prioritize_observations(observed_species2, priority_species2))
```

Expected Output:

```
["🐯", "🐯", "🐯", "🦌", "🐘", "🦁", "🦁", "🐻", "🦑", "🐼", "🐍"]
["cardinal", "sparrow", "bluejay", "crow", "robin"]
```

▼ ✨ **AI Hint: `extend()`**

Key Skill: Use AI to explain code concepts

This problem may benefit from use of the `extend()` function. For a quick refresher on how the `extend()` function works, check out the Unit 2 Cheatsheet.

If you'd still like to see more examples or ask follow-up questions, try using an AI tool like ChatGPT or GitHub Copilot. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Please provide 2-3 examples of how the `extend()` function is used in Python, and explain how each one works."

Problem 5: Calculating Conservation Statistics

You are given a 0-indexed integer array `species_populations` of even length, where each element represents the population of a particular species in a wildlife reserve.

As long as `species_populations` is not empty, you must repetitively:

1. Find the species with the minimum population and remove it.
2. Find the species with the maximum population and remove it.
3. Calculate the average population of the two removed species.

The average of two numbers `a` and `b` is `(a+b)/2`.

For example, the average of `200` and `300` is `(200+300)/2=250`.

Return the number of distinct averages calculated using the above process.

Note that when there is a tie for a minimum or maximum population, any can be removed.

```python
def distinct_averages(species_populations):
  pass
```

Example Usage:

```python
species_populations1 = [4,1,4,0,3,5]
species_populations2 = [1,100]

print(distinct_averages(species_populations1))
print(distinct_averages(species_populations2))
```

Example Output:

```
2
Example 1 Explanation:
1. Remove 0 and 5, and the average is (0 + 5) / 2 = 2.5. Now, nums = [4,1,4,3].
2. Remove 1 and 4. The average is (1 + 4) / 2 = 2.5, and nums = [4,3].
3. Remove 3 and 4, and the average is (3 + 4) / 2 = 3.5.
Since there are 2 distinct numbers among 2.5, 2.5, and 3.5, we return 2.

1
Example 2 Explanation:
There is only one average to be calculated after removing 1 and 100,
so we return 1.
```

Problem 6: Wildlife Reintroduction

As a conservationist, your research center has been raising multiple endangered species and is now ready to reintroduce them into their native habitats. You are given two 0-indexed strings `raised_species` and `target_species`. The string `raised_species` represents the list of species available to release into the wild at your center, where each character represents a different species. The string `target_species` represents a specific sequence of species you want to form and release together.

You can take some species from `raised_species` and rearrange them to form new sequences.

Return the maximum number of copies of `target_species` that can be formed by taking species from `raised_species` and rearranging them.

```python
def max_species_copies(raised_species, target_species):
  pass
```

Example Usage:

```python
raised_species1 = "abcba"
target_species1 = "abc"
print(max_species_copies(raised_species1, target_species1))  # Output: 1

raised_species2 = "aaaaabbbbcc"
target_species2 = "abc"
print(max_species_copies(raised_species2, target_species2)) # Output: 2
```

Example Output:

```
1
Example 1 Explanation:
We can make one copy of "abc" by taking the letters at indices 0, 1, and 2.
We can make at most one copy of "abc", so we return 1.
Note that while there is an extra 'a' and 'b' at indices 3 and 4, we cannot
reuse the letter 'c' at index 2, so we cannot make a second copy of "abc".

2
Example 2 Explanation:
We can make one copy of "abc" by taking the letters at indices 0, 5, and 9.
We can make a second copy of "abc" by taking the letters at indices 1, 6, and 10
At this point we are out of the letter "c" and cannot make additional copies.
```

Problem 7: Count Unique Species

You are given a string `ecosystem_data` that consists of digits and lowercase English letters. The digits represent the observed counts of various species in a protected ecosystem.

You will replace every non-digit character with a space. For example, `"f123de34g8hi34"` will become `" 123 34 8 34"`. Notice that you are left with some species counts that are separated by at least one space: `"123", "34", "8", and "34"`.

Return the number of unique species counts after performing the replacement operations on `ecosystem_data`.

Two species counts are considered different if their decimal representations without any leading zeros are different.

```python
def count_unique_species(ecosystem_data):
  pass
```

Example Usage:

```
ecosystem_data1 = "f123de34g8hi34"
ecosystem_data2 = "species1234forest234"
ecosystem_data3 = "x1y01z001"

print(count_unique_species(ecosystem_data1))
print(count_unique_species(ecosystem_data2))
print(count_unique_species(ecosystem_data3))
```

Example Output:

```
3
2
1
```

▼ ✨ **AI Hint: Representing Infinite Values**

Key Skill: Use AI to explain code concepts

To solve this problem, it may be helpful to know how to represent **positive or negative infinity** in Python. TO learn more, take a look at the Infinity section of the Unit 9 Cheatsheet.

If you still have questions, try asking an AI tool like ChatGPT or GitHub Copilot to explain more about positive and negative infinity. For example, you might ask:

"What is a common use case for positive or negative infinity in a program?"

Problem 8: Equivalent Species Pairs

In an effort to understand species diversity in different habitats, researchers are analyzing species pairs observed in various regions. Each pair is represented by a list `[a, b]` where `a` and `b` represent two species observed together.

A species pair `[a, b]` is considered equivalent to another pair `[c, d]` if and only if either `(a == c and b == d)` or `(a == d and b == c)`. This means that the order of species in a pair does not matter.

Your task is to determine the number of equivalent species pairs in the list of observed species pairs.

```
def num_equiv_species_pairs(species_pairs):
  pass
```

Example Usage:

```
species_pairs1 = [[1,2],[2,1],[3,4],[5,6]]
species_pairs2 = [[1,2],[1,2],[1,1],[1,2],[2,2]]

print(num_equiv_species_pairs(species_pairs1))
print(num_equiv_species_pairs(species_pairs2))
```

Example Output:

```
1
3
```

▼ 💡 **Hint: Calculating the Number of Equivalent Pairs**

For a species pair that appears `n` times, the number of equivalent pairs that can be formed is given by the formula the formula : `c * (c - 1) // 2`

Close Section

▼ Standard Problem Set Version 2

Problem 1: Filter Destinations

You're planning an epic trip and have a dictionary of destinations mapped to their respective rating scores. Your goal is to visit only the best-rated destinations. Write a function that takes in a dictionary `destinations` and a `rating_threshold` as parameters. The function should iterate through the dictionary and remove all destinations that have a rating strictly below the `rating_threshold`. Return the updated dictionary.

```
def remove_low_rated_destinations(destinations, rating_threshold):
  pass
```

Example Usage:

```
destinations = {"Paris": 4.8, "Berlin": 3.5, "Addis Ababa": 4.9, "Moscow": 2.8}
destinations2 = {"Bogotá": 4.8, "Kansas City": 3.9, "Tokyo": 4.5, "Sydney": 3.0}

print(remove_low_rated_destinations(destinations, 4.0))
print(remove_low_rated_destinations(destinations2, 4.9))
```

Example Output:

```
{"Paris": 4.8, "Addis Ababa": 4.9}
{}
```

▼ ✨ AI Hint: Accessing Values in a Dictionary

Key Skill: Use AI to explain code concepts

This question will require you to use keys to access their corresponding values in a dictionary. There are two common ways to access values in a dictionary. Try asking ChatGPT or GitHub copilot:

"You're an expert computer science tutor. Please show me the two most common ways to access values in a dictionary in Python, and explain how each one works."

Then open the next hint to see the answer!

▼ 💡 **Hint: Dictionary Access options**

The two common ways to access values in a dictionary are square bracket notation `d[key]` and the `get()` method.

The Unit 2 cheatsheet includes a more thorough breakdown of these two options. If you still feel confused after reviewing the cheatsheet, try asking generative AI to help you understand!

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

This question will require you to loop over a dictionary. We have three options for looping over a dictionary: looping over the keys, values, or key-value pairs. To explore how to access the keys, values, and key-value pairs reference the unit cheatsheet. For specific examples of looping over a dictionary, ask a generative AI tool to provide an example or search for existing examples using a search engine.

Problem 2: Unique Travel Souvenirs

As a seasoned traveler, you've collected a variety of souvenirs from different destinations. You have an array of string `souvenirs`, where each string represents a type of souvenir. You want to know if the number of occurrences of each type of souvenir in your collection is unique.

Write a function that takes in an array `souvenirs` and returns `True` if the number of occurrences of each value in the array is unique, or `False` otherwise.

```
def unique_souvenir_counts(souvenirs):
  pass
```

Example Usage:

```
souvenirs1 = ["keychain", "hat", "hat", "keychain", "keychain", "postcard"]
souvenirs2 = ["postcard", "postcard", "postcard", "postcard"]
souvenirs3 = ["keychain", "magnet", "hat", "candy", "postcard", "stuffed bear"]

print(unique_souvenir_counts(souvenirs1))
print(unique_souvenir_counts(souvenirs2))
print(unique_souvenir_counts(souvenirs3))
```

Example Output:

```
True
Example 1 Explanation: The value "keychain" has 3 occurrences, "hat" has 2
and "postcard" has 1. No two values have the same number of occurrences.

True
Example 2 Explanation: The value "postcard" appears 4 times There's only one count (4), whicl

False
Example 3 Explanation: Each item appears 1 time All counts are 1, which is not unique, so th:
```

▼ ✨ **AI Hint: Introduction to sets**

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **set**. A Python set is a data type which holds an unordered, mutable collection of *unique* elements.

If you are unfamiliar with what a set is, or how to create a set, you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a set is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a set different from a list or dictionary? Can you show me examples of each?"

▼ ✨ **AI Hint: Frequency Maps**

Key Skill: Use AI to explain code concepts

A dictionary that maps unique values to their frequencies within a given data structure or data type is often called a **frequency map**. Frequency maps are an extremely useful problem solving tool that you will see often throughout this unit and in future units.

We encourage you to learn by doing and attempt this problem before doing a deeper dive! However, if you get stuck, you can ask a generative AI tool like ChatGPT or GitHub Copilot to explain the concept.

For example, you could say:

"You're an expert computer science tutor for a Python-based technical interviewing course. Please explain what a frequency map is, and provide one or more examples of simple technical interview problems in which a frequency map is useful."

Problem 3: Secret Beach

You make friends with a local at your latest destination, and they give you a coded message with the name of a secret beach most tourists don't know about! You are given the strings `key` and `message` which represent a cipher key and a secret message, respectively. The steps to decode the message are as follows:

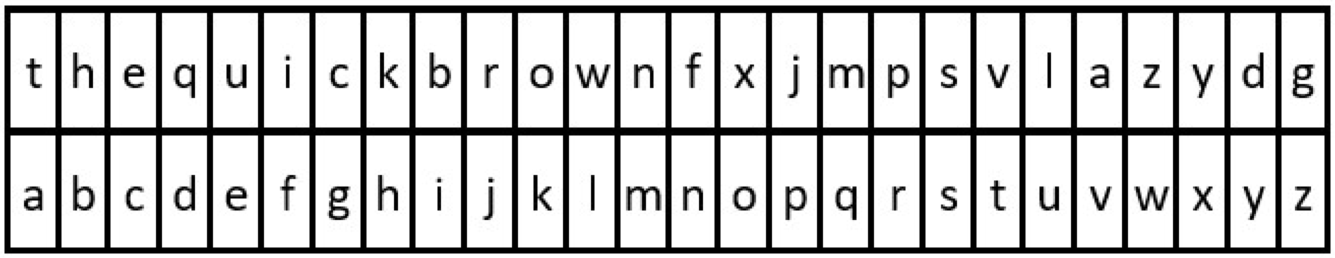
1. Use the first appearance of all 26 lowercase English letters in key as the order of the substitution table.
2. Align the substitution table with the regular English alphabet.
3. Each letter in message is then substituted using the table.
4. Spaces `' '` are transformed to themselves.

For example, given `key = "travel the world"` (an actual key would have at least one instance of each letter in the alphabet), we have the partial substitution table of `('t' -> 'a', 'r' -> 'b', 'a' -> 'c', 'v' -> 'd', 'e' -> 'e', 'l' -> 'f', 'h' -> 'g', 'w' -> 'h'` .

Write a function `decode_message()` that accepts the strings `key` and `message` and returns a string representing the decoded message.

```
def decode_message(key, message):
  pass
```

Example Usage 1:



t	h	e	q	u	i	c	k	b	r	o	w	n	f	x	j	m	p	s	v	l	a	z	y	d	g
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

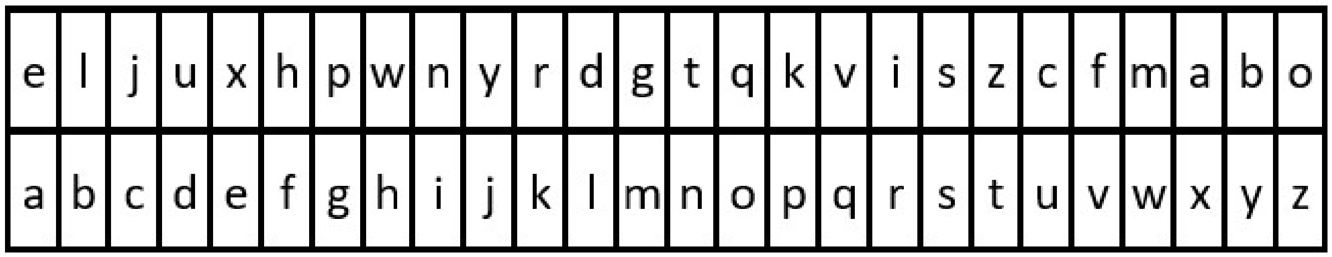
```
key1 = "the quick brown fox jumps over the lazy dog"
message1 = "vkbs bs t suepuv"

print(decode_message(key1, message1))
```

Example Output 1:

```
this is a secret
```

Example Usage 2:



e	l	j	u	x	h	p	w	n	y	r	d	g	t	q	k	v	i	s	z	c	f	m	a	b	o
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

```
key2 = "eljuxhpwnyrdgtqkviszcfmabo"
message2 = "hntu depcte lxejw lxwntu zwx piqfx"

print(decode_message(key2, message2))
```

Example Output 2:

```
find laguna beach behind the grove
```

Problem 4: Longest Harmonious Travel Sequence

In a list of travel packages, we define a harmonious travel sequence as a sequence where the difference between the maximum and minimum travel ratings is exactly 1.

Given an integer array `rating`, return the length of the longest harmonious travel sequence among all its possible subsequences.

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

You are provided with a partially implemented solution that contains bugs. Your task is to identify and fix the bugs to ensure the solution works correctly.

```
def find_longest_harmonious_travel_sequence(ratings):
    # Initialize a dictionary to store the frequency of each rating
    frequency = {}

    # Count the occurrences of each rating
    for rating in ratings:
        frequency[rating] += 1

    max_length = 0

    # Find the longest harmonious sequence
    for rating in frequency:
        if rating + 1 in frequency:
            max_length = max(max_length,
                        frequency[rating] + frequency[rating - 1])

    return max_length
```

Example Usage:

```
durations1 = [1, 3, 2, 2, 5, 2, 3, 7]
durations2 = [1, 2, 3, 4]
durations3 = [1, 1, 1, 1]

print(find_longest_harmonious_travel_sequence(durations1))
print(find_longest_harmonious_travel_sequence(durations2))
print(find_longest_harmonious_travel_sequence(durations3))
```

Example Output:

```
5
2
0
```

Problem 5: Check if All Destinations in a Route are Covered

You are given a 2D integer array `trips` and two integers `start_dest` and `end_dest`. Each `trips[i] = [starti, endi]` represents an inclusive travel interval between `starti` and `endi`.

Return `True` if each destination in the inclusive route `[start_dest, end_dest]` is covered by at least one trip in `trips`. Return `False` otherwise.

A destination `x` is covered by a trip `trips[i] = [starti, endi]` if `starti <= x <= endi`.

```
def is_route_covered(trips, start_dest, end_dest):
  pass
```

Example Usage:

```
trips1 = [[1, 2], [3, 4], [5, 6]]
start_dest1, end_dest1 = 2, 5

trips2 = [[1, 10], [10, 20]]
start_dest2, end_dest2 = 21, 21

trips3 = [[1, 2], [3, 5]]
start_dest3, end_dest1 = 2, 5

print(is_route_covered(trips1, start_dest1, end_dest1))
print(is_route_covered(trips2, start_dest2, end_dest2))
print(is_route_covered(trips3, start_dest3, end_dest3))
```

Example Output:

```
True
False
True
```

Problem 6: Most Popular Even Destination

Given a list of integers `destinations`, where each integer represents the popularity score of a destination, return the most popular even destination.

If there is a tie, return the smallest one. If there is no such destination, return `-1`.

```
def most_popular_even_destination(destinations):
  pass
```

Example Usage:

```
destinations1 = [0, 1, 2, 2, 4, 4, 1]
destinations2 = [4, 4, 4, 9, 2, 4]
destinations3 = [29, 47, 21, 41, 13, 37, 25, 7]

print(most_popular_even_destination(destinations1))
print(most_popular_even_destination(destinations2))
print(most_popular_even_destination(destinations3))
```

Example Output:

```
2
4
-1
```

Problem 7: Check if Itinerary is Valid

You are given an itinerary `itinerary` representing a list of trips between cities, where each city is represented by an integer. We consider an itinerary valid if it is a permutation of an itinerary template `base[n]`.

The template `base[n]` is defined as `[1, 2, ..., n - 1, n, n]` (in other words, it is an itinerary of length `n + 1` that visits cities `1` to `n - 1` exactly once, plus visits city `n` twice). For example, `base[1] = [1, 1]` and `base[3] = [1, 2, 3, 3]`.

Return `True` if the given itinerary is valid, otherwise return `False`.

A **permutation** is an arrangement of a set of elements. For example `[3, 2, 1]` and `[2, 3, 1]` are both possible permutations of the set of numbers `1`, `2`, and `3`.

```
def is_valid_itinerary(itinerary):
  pass
```

Example Usage:

```
itinerary1 = [2, 1, 3]
itinerary2 = [1, 3, 3, 2]
itinerary3 = [1, 1]

print(is_valid_itinerary(itinerary1))
print(is_valid_itinerary(itinerary2))
print(is_valid_itinerary(itinerary3))
```

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 itinerary1 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 itinerary2 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 itinerary3 is a permutation of base[1] = [1, 1]. Therefore, the
 answer is true.
```

Problem 8: Finding Common Tourist Attractions with Least Travel Time

Given two lists of tourist attractions, `tourist_list1` and `tourist_list2`, find the common attractions with the least total travel time.

A common attraction is one that appears in both `tourist_list1` and `tourist_list2`.

A common attraction with the least total travel time is a common attraction such that if it appeared at `tourist_list1[i]` and `tourist_list2[j]` then `i + j` should be the minimum value among all the other common attractions.

Return all the common attractions with the least total travel time. Return the answer in any order.

```python
def find_attractions(tourist_list1, tourist_list2):
  pass
```

Example Usage:

```python
tourist_list1 = ["Eiffel Tower","Louvre Museum","Notre-Dame","Disneyland"]
tourist_list2 = ["Colosseum","Trevi Fountain","Pantheon","Eiffel Tower"]

print(find_attractions(tourist_list1, tourist_list2))

tourist_list1 = ["Eiffel Tower","Louvre Museum","Notre-Dame","Disneyland"]
tourist_list2 = ["Disneyland","Eiffel Tower","Notre-Dame"]

print(find_attractions(tourist_list1, tourist_list2))

tourist_list1 = ["beach","mountain","forest"]
tourist_list2 = ["mountain","beach","forest"]

print(find_attractions(tourist_list1, tourist_list2))
```

Example Output:

```
["Eiffel Tower"]
["Eiffel Tower"]
["mountain", "beach"]
```

▼ ✨ **AI Hint: Representing Infinite Values**

Key Skill: Use AI to explain code concepts

To solve this problem, it may be helpful to know how to represent **positive or negative infinity** in Python. TO learn more, take a look at the Infinity section of the Unit 9 Cheatsheet.

If you still have questions, try asking an AI tool like ChatGPT or GitHub Copilot to explain more about positive and negative infinity. For example, you might ask:

"What is a common use case for positive or negative infinity in a program?"

Close Section

▾ 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.

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

Example Usage:

```python
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.org", "5
"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.

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

Example Usage:

```python
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
```

▼ ✨ **AI Hint: Representing Infinite Values**

Key Skill: Use AI to explain code concepts

To solve this problem, it may be helpful to know how to represent **positive or negative infinity** in Python. TO learn more, take a look at the Infinity section of the Unit 9 Cheatsheet.

If you still have questions, try asking an AI tool like ChatGPT or GitHub Copilot to explain more about positive and negative infinity. For example, you might ask:

"What is a common use case for positive or negative infinity in a program?"

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.

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

Example Input:

```python
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.

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

Example Usage:

```
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