TIP102 | Intermediate Technical Interview Prep

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

Session 1: Recursion

Session Overview

In this session, students will dive deep into the concept of recursion, a fundamental programming technique used to solve problems by breaking them down into simpler, self-similar subproblems. The session will cover how to write recursive functions - specifically how to identify the base case and the importance of recursive calls, equipping students with the skills needed to tackle recursive programming questions.

> 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**

Please Note: **It is not required or expected** that you complete all of the practice problems! In some sessions you may only complete 1 problem and that's okay.

Strengthening your **approach** to problems, and your **ability to speak and engage through the process** are key skills most often underdeveloped for engineers at this stage - focus on those in our small groups for your long term success!

You can always return to problems independently, after class time, to embrace the technical concepts and gain additional practice.

Close Section

Problem Solving Approach

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

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

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

We will learn to:

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

Breakout Problems Session 1

Standard Problem Set Version 1

Problem 1: Counting Iron Man's Suits

Tony Stark, aka Iron Man, has designed many different suits over the years. Given a list of strings `suits` where each string is a suit in Stark's collection, count the total number of suits in the list.

1. Implement the solution *iteratively* without the use of the `len()` function.
2. Implement the solution *recursively*.
3. Discuss: what are the similarities between the two solutions? What are the differences?

```
def count_suits_iterative(suits):
    pass

def count_suits_recursive(suits):
    pass
```

Example Usage:

```
print(count_suits_iterative(["Mark I", "Mark II", "Mark III"]))
print(count_suits_recursive(["Mark I", "Mark I", "Mark III", "Mark IV"]))
```

Example Output:

```
3
4
```

▼ 💡 **Hint: Recursion**

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

Problem 2: Collecting Infinity Stones

Thanos is collecting Infinity Stones. Given an array of integers `stones` representing the power of each stone, return the total power using a recursive approach.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
def sum_stones(stones):
    pass
```

Example Usage:

```
print(sum_stones([5, 10, 15, 20, 25, 30]))
print(sum_stones([12, 8, 22, 16, 10]))
```

Example Output:

```
105
68
```

Problem 3: Counting Unique Suits

Some of Iron Man's suits are duplicates. Given a list of strings `suits` where each string is a suit in Stark's collection, count the total number of *unique* suits in the list.

1. Implement the solution *iteratively*.
2. Implement the solution *recursively*.
3. Discuss: what are the similarities between the two solutions? What are the differences?
4. Evaluate the time complexity of each solution. Are they the same? Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
def count_suits_iterative(suits):
    pass

def count_suits_recursive(suits):
    pass
```

Example Usage:

```
print(count_suits_iterative(["Mark I", "Mark II", "Mark III"]))
print(count_suits_recursive(["Mark I", "Mark I", "Mark III"]))
```

Example Output:

```
3
2
```

▼ 💡 **Hint: Multiple Recursive Cases**

This problem has multiple recursive cases! To see an example of a function with multiple recursive cases, check out the Building a Recursive Function section of the unit cheatsheet.

Problem 4: Calculating Groot's Growth

Groot grows according to a pattern similar to the Fibonacci sequence. Given `n`, find the height of Groot after `n` months using a recursive method.

The Fibonacci numbers, commonly denoted `F(n)` form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from `0` and `1`. That is,

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def fibonacci_growth(n):
    pass
```

Example Usage:

```python
print(fibonacci_growth(5))
print(fibonacci_growth(8))
```

Example Output:

```
5
21
```

Problem 5: Calculating the Power of the Fantastic Four

The superhero team, The Fantastic Four, are training to increase their power levels. Their power level is represented as a power of 4. Write a recursive function that calculates the power of 4 raised to the nth power to determine their training level.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def power_of_four(n):
    pass
```

Example Usage:

```python
print(power_of_four(2))
print(power_of_four(-2))
```

Example Output:

```
16
Example 1 Explanation: 2 to the 4th power (4 * 4) is 16.
16
Example 2 Explanation: -2 to the 4th power is 1/(4 * 4) is 0.0625.
```

Problem 6: Strongest Avenger

The Avengers need to determine who is the strongest. Given a list of their strengths, find the maximum strength using a recursive approach without using the `max()` function.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def strongest_avenger(strengths):
    pass
```

Example Usage:

```python
print(strongest_avenger([88, 92, 95, 99, 97, 100, 94]))
print(strongest_avenger([50, 75, 85, 60, 90]))
```

Example Output:

```
100
Example 1 Explanation: The maximum strength among the Avengers is 100.

90
Example 2 Explanation: The maximum strength among the Avengers is 90.
```

Problem 7: Counting Vibranium Deposits

In Wakanda, vibranium is the most precious resource, and it is found in several deposits. Each deposit is represented by a character in a string (e.g., `"V"` for vibranium, `"G"` for gold, etc.)

Given a string `resources`, write a recursive function `count_deposits()` that returns the total number of distinct *vibranium* deposits in `resources`.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def count_deposits(resources):
    pass
```

Example Usage:

```python
print(count_deposits("VVVVV"))
print(count_deposits("VXVYGA"))
```

Example Output:

```
5
2
Example 2 Explanation: There are two characters "V" in the string "VXVYGA",
therefore there are two vibranium deposits in the string.
```

Problem 8: Merging Missions

The Avengers are planning multiple missions, and each mission has a priority level represented as a node in a linked list. You are given the heads of two sorted linked lists, `mission1` and `mission2`, where each node represents a mission with its priority level.

Implement a recursive function `merge_missions()` which merges these two mission lists into one sorted list, ensuring that the combined list maintains the correct order of priorities. The merged list should be made by splicing together the nodes from the first two lists.

Return the head of the merged mission linked list.

```python
class Node:
  def __init__(self, value, next=None):
      self.value = value
      self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_missions(mission1, mission2):
    pass
```

Example Usage:

```python
mission1 = Node(1, Node(2, Node(4)))
mission2 = Node(1, Node(3, Node(4)))

print_linked_list(merge_missions(mission1, mission2))
```

```
1 -> 1 -> 2 -> 3 -> 4 -> 4
```

Problem 9: Merging Missions II

Below is an iterative solution to the `merge_missions()` function from the previous problem. Compare your recursive solution to the iterative solution below.

Discuss with your podmates. Which solution do you prefer?

```python
class Node:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_missions_iterative(mission1, mission2):
    temp = Node()  # Temporary node to simplify the merging process
    tail = temp

    while mission1 and mission2:
        if mission1.value < mission2.value:
            tail.next = mission1
            mission1 = mission1.next
        else:
            tail.next = mission2
            mission2 = mission2.next
        tail = tail.next

    # Attach the remaining nodes, if any
    if mission1:
        tail.next = mission1
    elif mission2:
        tail.next = mission2

    return temp.next  # Return the head of the merged linked list
```

Close Section

▾ Standard Problem Set Version 2

Problem 1: Calculating Village Size

In the kingdom of Codepathia, the queen determines how many resources to distribute to a village based on its class. A village's class is equal to the number of digits in its population. Given an integer `population`, write a function `get_village_class()` that returns the number of digits in `population`.

1. Implement the solution *iteratively*.
2. Implement the solution *recursively*.
3. Discuss: what are the similarities between the two solutions? What are the differences?

```
def get_village_class_iterative(population):
    pass

def get_village_class_recursive(population):
    pass
```

Example Usage:

```
print(get_village_class_iterative(432))
print(get_village_class_recursive(432))
print(get_village_class_iterative(9))
print(get_village_class_recursive(9))
```

Example Output:

```
3
3
1
1
```

▼ 💡 **Hint: Recursion**

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

Problem 2: Counting the Castle Walls

In a faraway kingdom, a castle is surrounded by multiple defensive walls, where each wall is nested within another. Given a list of lists `walls` where each list `[]` represents a wall, write a recursive function `count_walls()` that returns the total number of walls.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```
def count_walls(walls):
    pass
```

Example Usage:

```
walls = ["outer", ["inner", ["keep", []]]]

print(count_walls(walls))
print(count_walls([]))
```

Example Output:

```
4
1
```

Problem 3: Reversing a Scroll

A wizard is deciphering an ancient scroll and needs to reverse the letters in a word to reveal a hidden message. Write a recursive function to reverse the letters in a given `scroll` and returns the reversed `scroll`. Assume `scroll` only contains alphabetic characters.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def reverse_scroll(scroll):
    pass
```

Example Usage:

```python
print(reverse_scroll("cigam"))
print(reverse_scroll("lleps"))
```

Example Output:

```
magic
spell
```

Problem 4: Palindromic Name

Queen Ada is superstitious and believes her children will only have good fortune if their name is symmetrical and reads the same forward and backward. Write a recursive function that takes in a string comprised of only lowercase alphabetic characters `name` and returns `True` if the name is palindromic and `False` otherwise.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def is_palindrome(name):
    pass
```

Example Usage:

```python
print(is_palindrome("eve"))
print(is_palindrome("ling"))
print(is_palindrome(""))
```

Example Output:

```
True
True
False
```

▼ 💡 **Hint: Multiple Recursive Cases**

This problem has multiple recursive cases! To see an example of a function with multiple recursive cases, check out the Building a Recursive Function section of the unit cheatsheet.

Problem 5: Doubling the Power of a Spell

The court magician is practicing a spell that doubles its power with each incantation. Given an integer `initial_power` and a non-negative integer `n`, write a recursive function that doubles `initial_power` `n` times.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def double_power(initial_power, n):
    pass
```

Example Usage:

```python
print(double_power(5, 3))
print(double_power(7, 2))
```

Example Output

```
40
Example 1 Explanation: 5 doubled 3 times: 5 -> 10 -> 20 -> 40

Output: 28
Example 2 Explanation: 7 doubled 2 times: 7 -> 14 -> 28
```

Problem 6: Checking the Knight's Path

A knight is traveling along a path marked by stones, and each stone has a number on it. The knight must check if the numbers on the stones form a strictly increasing sequence. Write a recursive function to determine if the sequence is strictly increasing.

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def is_increasing_path(path):
    pass
```

Example Usage:

```python
print(is_increasing_path([1, 2, 3, 4, 5]))
print(is_increasing_path([3, 5, 2, 8]))
```

Example Output

```
True
False
```

Problem 7: Finding the Longest Winning Streak

In the kingdom's grand tournament, knights compete in a series of challenges. A knight's performance in the challenge is represented by a string `challenges`, where a success is represented by an `S` and each other outcome (like a draw or loss) is represented by an `"O"`. Write a recursive function to find the length of the longest consecutive streak of successful challenges (`"S"`).

Evaluate the time complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time complexity.

```python
def longest_streak(frames, current_length=0, max_length=0):
    pass
```

Example Usage:

```python
print(longest_streak("SSOSSS"))
print(longest_streak("SOSOSOSO"))
```

Example Output:

```
3
1
```

Problem 8: Weaving Spells

A magician can double a spell's power if they merge two incantations together. Given the heads of two linked lists `spell_a` and `spell_b` where each node in the lists contains a spell segment, write a recursive function `weave_spells()` that weaves spells in the pattern:

`a1 -> b1 -> a2 -> b2 -> a3 -> b3 -> ...`

```python
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def weave_spells(spell_a, spell_b)
    pass
```

Example Usage:

```python
spell_a = Node('A', Node('C', Node('E')))
spell_b = Node('B', Node('D', Node('F')))

print_linked_list(weave_spells(spell_a, spell_b))
```

Example Output:

```
A -> B -> C -> D -> E -> F
```

Problem 9: Weaving Spells II

Below is an iterative solution to the `weaving_spells()` function from the previous problem. Compare your recursive solution to the iterative solution below.

Discuss with your podmates. Which solution do you prefer?

```python
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def weave_spells(spell_a, spell_b):
    # If either list is empty, return the other
    if not spell_a:
        return spell_b
    if not spell_b:
        return spell_a

    # Start with the first node of spell_a
    head = spell_a

    # Loop through both lists until one is exhausted
    while spell_a and spell_b:
        # Store the next pointers
        next_a = spell_a.next
        next_b = spell_b.next

        # Weave spell_b after spell_a
        spell_a.next = spell_b

        # If there's more in spell_a, weave it after spell_b
        if next_a:
            spell_b.next = next_a

        # Move to the next nodes
        spell_a = next_a
        spell_b = next_b

    # Return the head of the new woven list
    return head
```

Close Section

▾ Advanced Problem Set Version 1

Problem 1: Counting the Layers of a Sandwich

You're working at a deli, and need to count the layers of a sandwich to make sure you made the order correctly. Each layer is represented by a nested list. Given a list of lists `sandwich` where each list `[]` represents a sandwich layer, write a recursive function `count_layers()` that returns the total number of sandwich layers.

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_layers(sandwich):
    pass
```

Example Usage:

```
sandwich1 = ["bread", ["lettuce", ["tomato", ["bread"]]]]
sandwich2 = ["bread", ["cheese", ["ham", ["mustard", ["bread"]]]]]

print(count_layers(sandwich1))
print(count_layers(sandwich2))
```

Example Output:

```
4
5
```

▼ 💡 **Hint: Recursion**

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

Problem 2: Reversing Deli Orders

The deli counter is busy, and orders have piled up. To serve the last customer first, you need to reverse the order of the deli orders. Given a string `orders` where each individual order is separated by a single space, write a recursive function `reverse_orders()` that returns a new string with the orders reversed.

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_orders(orders):
    pass
```

Example Usage:

```
print(reverse_orders("Bagel Sandwich Coffee"))
```

Example Output:

```
Coffee Sandwich Bagel
```

▼ 💡 **Hint: Recursive Helpers**

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

Problem 3: Sharing the Coffee

The deli staff is in desperate need of caffeine to keep them going through their shift and has decided to divide the coffee supply equally among themselves. Each batch of coffee is stored in containers of different sizes. Write a recursive function `can_split_coffee()` that accepts a list of integers `coffee` representing the volume of each batch of coffee and returns `True` if the coffee can be split evenly by volume among `n` staff 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 can_split_coffee(coffee, n):
pass
```

Example Usage:

```
print(can_split_coffee([4, 4, 8], 2))
print(can_split_coffee([5, 10, 15], 4))
```

Example Output:

```
True
False
```

Problem 4: Super Sandwich

A regular at the deli has requested a new order made by merging two different sandwiches on the menu together. Given the heads of two linked lists `sandwich_a` and `sandwich_b` where each node in the lists contains a spell segment, write a recursive function `merge_orders()` that merges the two sandwiches together in the pattern:

`a1 -> b1 -> a2 -> b2 -> a3 -> b3 -> ...`

Return the head of the merged sandwich.

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.

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_orders(sandwich_a, sandwich_b)
    pass
```

Example Usage:

```
sandwich_a = Node('Bacon', Node('Lettuce', Node('Tomato')))
sandwich_b = Node('Turkey', Node('Cheese', Node('Mayo')))
sandwich_c = Node('Bread')

print_linked_list(merge_orders(sandwich_a, sandwich_b))
print_linked_list(merge_orders(sandwich_a, sandwich_c))
```

Example Output:

```
Bacon -> Turkey -> Lettuce -> Cheese -> Tomato -> Mayo
Bacon -> Bread -> Lettuce -> Tomato
```

Problem 5: Super Sandwich II

Below is an iterative solution to the `merge_orders()` function from the previous problem. Compare your recursive solution to the iterative solution below.

Discuss with your podmates. Which solution do you prefer? How do they compare on time complexity? Space complexity?

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_orders(sandwich_a, sandwich_b):
    # If either list is empty, return the other
    if not sandwich_a:
        return sandwich_b
    if not sandwich_b:
        return sandwich_a

    # Start with the first node of sandwich_a
    head = sandwich_a

    # Loop through both lists until one is exhausted
    while sandwich_a and sandwich_b:
        # Store the next pointers
        next_a = sandwich_a.next
        next_b = sandwich_b.next

        # Merge sandwich_b after sandwich_a
        sandwich_a.next = sandwich_b

        # If there's more in sandwich_a, add it after sandwich_b
        if sandwich_a:
            sandwich_b.next = next_a

        # Move to the next nodes
        sandwich_a = next_a
        sandwich_b = next_b

    # Return the head of the new merged list
    return head
```

Problem 6: Ternary Expression

Given a string `expression` representing arbitrarily nested ternary expressions, evaluate the expression, and return its result as a string.

You can always assume that the given expression is valid and only contains digits, `'?'`, `':'`, `'T'`, and `'F'` where `'T'` is `True` and `'F'` is `False`. All the numbers in the expression are one-digit numbers (i.e., in the range `[0, 9]`).

Ternary expressions use the following syntax:

`condition ? true_choice : false_choice`

- `condition` is evaluate first and determines which choice to make.
    - `true_choice` is taken if `condition` evaluates to `True`
    - `false_choice` is taken if `condition` evaluates to `False`

The conditional expressions group right-to-left, and the result of the expression will always evaluate to either a digit, `'T'` or `'F'`.

We have provided an iterative solution that uses an explicit stack. Implement a recursive solution `evaluate_ternary_expression_recursive()`.

```python
def evaluate_ternary_expression_iterative(expression):
    stack = []

    # Traverse the expression from right to left
    for i in range(len(expression) - 1, -1, -1):
        char = expression[i]

        if stack and stack[-1] == '?':
            stack.pop()  # Remove the '?'
            true_expr = stack.pop()  # True expression
            stack.pop()  # Remove the ':'
            false_expr = stack.pop()  # False expression

            if char == 'T':
                stack.append(true_expr)
            else:
                stack.append(false_expr)
        else:
            stack.append(char)

    return stack[0]

def evaluate_ternary_expression_recursive(expression):
    pass
```

Example Usage:

```python
print(evaluate_ternary_expression_recursive("T?2:3"))
print(evaluate_ternary_expression_recursive("F?1:T?4:5"))
print(evaluate_ternary_expression_recursive("T?T?F:5:3"))
```

Example Output:

```
2
Example 1 Explanation: If True, then result is 2; otherwise result is 3.

4
Example Explanation: The conditional expressions group right-to-left. Using parentheses,
it is read/evaluated as:
"(F ? 1 : (T ? 4 : 5))" --> "(F ? 1 : 4)" --> "4"
or "(F ? 1 : (T ? 4 : 5))" --> "(T ? 4 : 5)" --> "4"

F
Explanation: The conditional expressions group right-to-left. Using parentheses,
it is read/evaluated as:
"(T ? (T ? F : 5) : 3)" --> "(T ? F : 3)" --> "F"
"(T ? (T ? F : 5) : 3)" --> "(T ? F : 5)" --> "F"
```

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Mapping Atlantis' Hidden Chambers

Poseidon, the ruler of Atlantis, has a map that shows various chambers hidden deep beneath the ocean. The map is currently stored as a nested list `sections`, with each section containing smaller subsections. Write a recursive function `map_chambers()` that converts the map into a nested dictionary, where each section and subsection is a key-value pair.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```python
def map_chambers(sections):
    pass
```

Example Usage:

```python
sections = ["Atlantis", ["Coral Cave", ["Pearl Chamber"]]]
print(map_chambers(sections))
```

Example Output

```
{'Atlantis': {'Coral Cave': 'Pearl Chamber'}}
```

▼ 💡 Hint: Recursion

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

Problem 2: Finding the Longest Sequence of Trident Gems

The people of Atlantis are collecting rare Trident Gems as they explore the ocean. The gems are arranged in a sequence of integers representing their value. Write a recursive function that returns the length of the consecutive sequence of gems where each subsequent value increases by exactly 1.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```python
def longest_trident_sequence(gems):
    pass
```

Example Usage:

```python
print(longest_trident_sequence([1, 2, 3, 2, 3, 4, 5, 6]))
print(longest_trident_sequence([5, 10, 7, 8, 1, 2]))
```

Example Output:

```
5
Example 1 Explanation: longest sequence is 2, 3, 4, 5, 6

2
Example 2 Explanation: longest sequence is 7, 8 or 1, 2
```

▼ 💡 **Hint: Recursive Helpers**

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

Problem 3: Last Building Standing

In Atlantis, buildings are arranged in concentric circles. The Greek gods have become unhappy with Atlantis, and have decided to punish the city by sending floods to sink certain buildings into the ocean.

Assume there are `n` buildings in a circle numbered from `1` to `n` in clockwise order. More formally, moving clockwise from the `ith` building brings you the the `(i+1)th` building for `1 <= i < n`, and moving clockwise from the `nth` building brings you to the `1st` building.

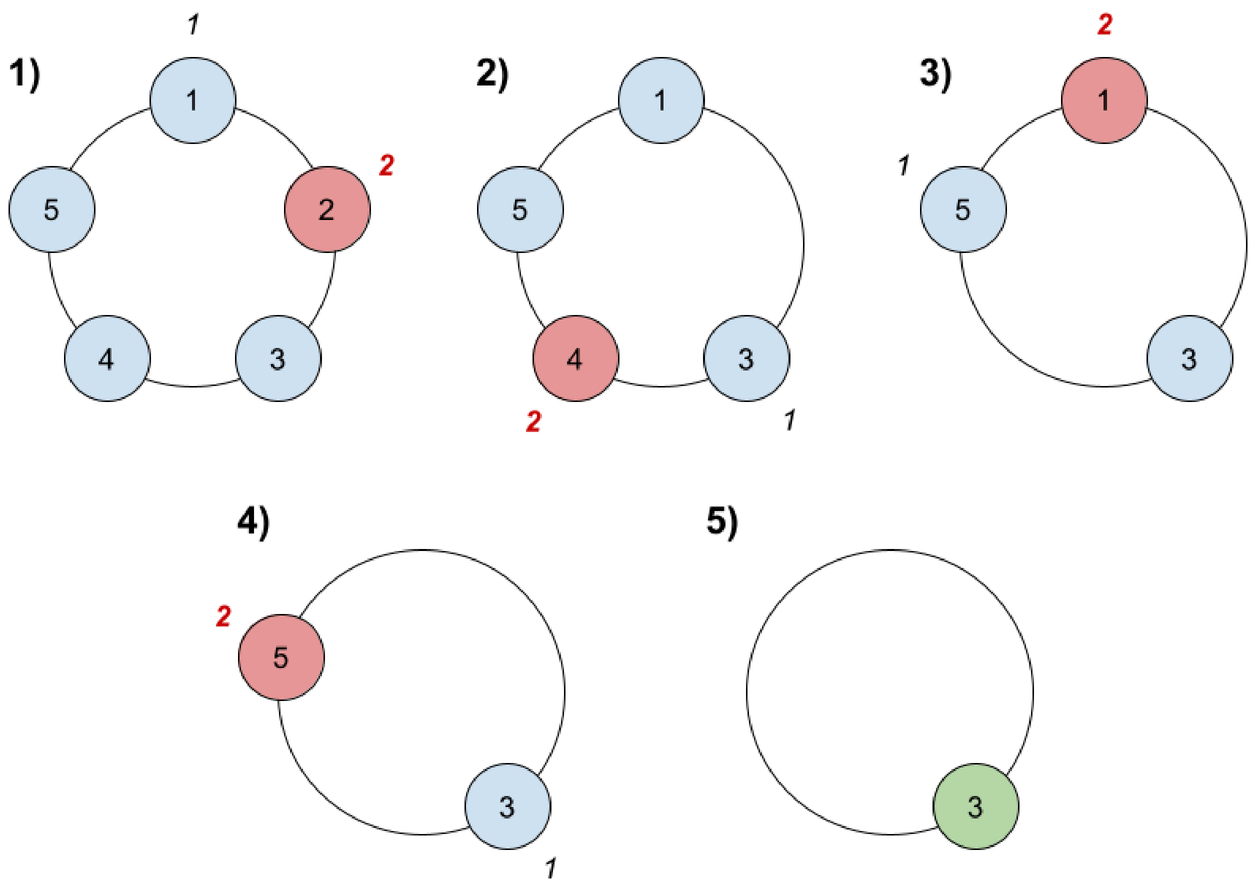
The gods are sinking buildings as follows:

1. Start with the `1st` building.
2. Count the next `k` buildings in the clockwise direction **including** the building you started at. The counting wraps around the circle and may count some buildings more than once.
3. The last building counted sinks and is removed from the circle.
4. If there is still more than one building standing in the circle, go back to step `2` **starting** from the building **immediately clockwise** of the building that was just sunk and repeat.
5. Otherwise, return the last building standing.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```python
def find_last_building(n, k):
    pass
```

Example Usage:



```python
print(find_last_building(5, 2))
print(find_last_building(6, 5))
```

Example Output:

```
3
Example 1 Explanation:
1) Start at building 1.
2) Count 2 buildings clockwise, which are buildings 1 and 2.
3) Building 2 sinks. Next start is building 3.
4) Count 2 buildings clockwise, which are buildings 3 and 4.
5) Building 4 sinks. Next start is building 5.
6) Count 2 buildings clockwise, which are buildings 5 and 1.
7) Building 1 sinks. Next start is building 3.
8) Count 2 buildings clockwise, which are buildings 3 and 5.
9) Building 5 sinks. Only building 3 is left, so they are the last building standing.

1
Example 2 Explanation:
Buildings sink in this order: 5, 4, 6, 2, 3. The last building is building 1.
```

Problem 4: Merging Missions

Atlanteans are planning multiple missions to explore the deep ocean, and each mission has a priority level represented as a node in a linked list. You are given the heads of two sorted linked lists, `mission1` and `mission2`, where each node represents a mission with its priority level.

Implement a recursive function `merge_missions()` which merges these two mission lists into one sorted list, ensuring that the combined list maintains the correct order of priorities. The merged list should be made by splicing together the nodes from the first two lists.

Return the head of the merged mission linked list.

```python
class Node:
  def __init__(self, value, next=None):
      self.value = value
      self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_missions(mission1, mission2):
    pass
```

Example Usage:

```python
mission1 = Node(1, Node(2, Node(4)))
mission2 = Node(1, Node(3, Node(4)))

print_linked_list(merge_missions(mission1, mission2))
```

```
1 -> 1 -> 2 -> 3 -> 4 -> 4
```

Problem 5: Merging Missions II

Below is an iterative solution to the `merge_missions()` function from the previous problem. Compare your recursive solution to the iterative solution below.

Discuss with your podmates. Which solution do you prefer? Which has better time complexity? Space complexity?

```python
class Node:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_missions_iterative(mission1, mission2):
    temp = Node()  # Temporary node to simplify the merging process
    tail = temp

    while mission1 and mission2:
        if mission1.value < mission2.value:
            tail.next = mission1
            mission1 = mission1.next
        else:
            tail.next = mission2
            mission2 = mission2.next
        tail = tail.next

    # Attach the remaining nodes, if any
    if mission1:
        tail.next = mission1
    elif mission2:
        tail.next = mission2

    return temp.next  # Return the head of the merged linked list
```

Problem 6: Decoding Ancient Atlantean Scrolls

In the mystical city of Atlantis, ancient scrolls have been discovered that contain encoded messages. These messages follow a specific encoding rule: `k[encoded_message]`, where the encoded_message inside the square brackets is repeated exactly `k` times. Note that `k` is guaranteed to be a positive integer.

You may assume that the input string `scroll` is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, `k`. For example, there will not be input like `3a` or `2[4]`. Your task is to decode these messages to reveal their original form.

We have provided an iterative solution that uses a stack. Write a function `decode_scroll_recursive()` that provides a recursive solution.

```
def decode_scroll(scroll):
    stack = []
    current_string = ""
    current_num = 0

    for char in scroll:
        if char.isdigit():
            # Build the number (could be more than one digit)
            current_num = current_num * 10 + int(char)
        elif char == '[':
            # Push the current number and current string to the stack
            stack.append((current_string, current_num))
            # Reset the current string and number
            current_string = ""
            current_num = 0
        elif char == ']':
            # Pop the last string and number from the stack
            prev_string, num = stack.pop()
            # Repeat the current string num times and add it to the previous string
            current_string = prev_string + current_string * num
        else:
            # Regular character, just add it to the current string
            current_string += char

    return current_string

def decode_scroll_recursive(scroll):
    pass
```

Example Usage:

```
scroll = "3[Coral2[Shell]]"
print(decode_scroll(scroll))

scroll = "2[Poseidon3[Sea]]"
print(decode_scroll(scroll))
```

Example Output:

```
CoralShellShellCoralShellShellCoralShellShell
PoseidonSeaSeaSeaPoseidonSeaSeaSea
```

Close Section