TIP102 | Intermediate Technical Interview Prep

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

Session 1: 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**

🔎 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 1

▶ Standard Problem Set Version 1

▶ Standard Problem Set Version 2

▼ 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.

```python
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**

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**

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?

```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_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 parenthe
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**

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.

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

Example Usage:

```
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**

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:

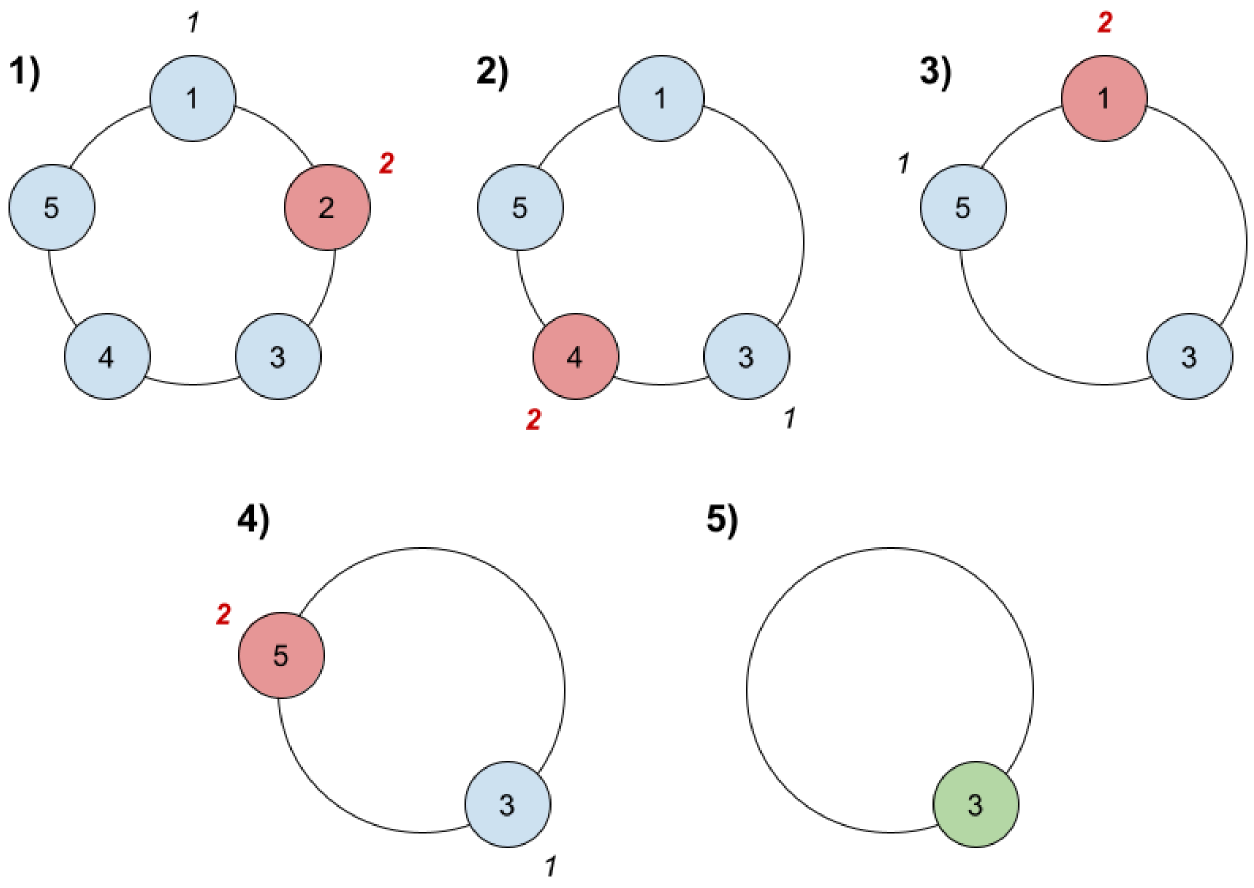
1. Start with the `1st` building.
2. Count the next `k` buildings in the clockwise direction **including** the building you started at. The counting wraps around the circle and may count some buildings more than once.
3. The last building counted sinks and is removed from the circle.
4. If there is still 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?

```
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