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

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**

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**

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.

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

Example Usage:

```
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.

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

Example Usage:

```
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.

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

Example Usage:

```
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.

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

Example Usage:

```
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.

```
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:

```
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?

```
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**

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.

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

Example Usage:

```
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.

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

Example Usage:

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

Example Output:

```
True
True
False
```

▶ 💡 **Hint: Multiple Recursive Cases**

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.

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

Example Usage:

```
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.

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

Example Usage:

```
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 `"0"`. 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.

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

Example Usage:

```
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 -> ...`

```
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:

```
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

▸ Advanced Problem Set Version 2