CodePath Courses

TIP101 | Intro to Technical Interview Prep

Intro to Technical Interview Prep Fall 2025 (@ Section 3 | Tuesdays and Thursdays 5PM - 7PM PT)
Personal Member ID#: **134071**

Need help? Post on our [class slack channel]() or email us at [support@codepath.org]()

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 the session deck, session recording, and more on the resources tab

🎢 Part 1 : Instructor Lead 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

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

💡 Unit 7 Cheatsheet

To help your learning journey with recursion and divide and conquer algorithms, we've put together a guide to common concepts and syntax you will use throughout Unit 7 breakout problems. Use this cheatsheet as a quick reference guide as you work through the problems below.

▾ Problem Set Version 1

Problem 1: Hello Hello

A recursive function is a function that calls itself within the body of the function.

Step 1: Copy the recursive function `repeat_hello()` into your IDE and run it.

Step 2: Then create another function `repeat_hello_iterative()` that produces the same output without using recursion.

Compare your iterative (non-recursive) solution to the recursive solution provided. What is similar? What is different?

```
                print("Hello")
                repeat_hello(n - 1)

repeat_hello(5)
```

▶ 💡 **Hint: Recursion**

Problem 2: Factorial Cases

Given the base case and recursive case, write a function `factorial()` that returns the factorial of a non-negative integer `n`. The factorial of a number is the product of all numbers between 1 and `n`.

Base Case: The smallest number we can find a factorial of is `0`. By definition, the factorial of `0` is `1`.

Recursive Case: We can restate the problem to say that the factorial of `n` is `n` * the factorial of `n-1`.

```
def factorial(n):
        pass
```

Example Usage:

```
# Example Input: 5
```

Example Output:

```
# Expected Output: 120
# Explanation: 5! = 5 * 4 * 3 * 2 * 1 = 120
```

▶ 💡 **Hint: Writing Recursive Functions**

Problem 3: Recursive Sum

Without using the built-in `sum()` function, write a function `sum_list()` that calculates the sum of all values in a list recursively.

What is the time complexity of this function? What is the space complexity?

```
def sum_list(lst):
        pass
```

Example Usage:

Example Output:

```
# Expected Output: 15
# Explanation: 1 + 2 + 3 + 4 + 5 = 15
```

Problem 4: Recursive Power of 2

Given an integer `n`, return `True` if `n` is a power of two. Otherwise, return `False``.

An integer `n` is a power of two if there exists an integer `x` such that `n == 2ˣ`.

Solve the problem recursively. What is the time complexity of this function? What is the space complexity?

```
def is_power_of_two(n):
        pass
```

Example Usage:

```
print(is_power_of_two(16))
print(is_power_of_two(18))
```

Example Output:

```
True
False
```

Problem 5: Binary Search I

Binary search is a searching algorithm that allows us to efficiently find the index of a given value within a sorted list. Given the pseudo code for binary search below, implement an *iterative* (non-recursive) implementation of binary search. There is also a recursive alternative that we'll cover in the session 2 problem set!

Evaluate the time and space complexity of your implementation.

```
def binary_search(lst, target):
        # Initialize a left pointer to the 0th index in the list
        # Initialize a right pointer to the last index in the list

        # While left pointer is less than right pointer:
                # Find the middle index of the array

                # If the value at the middle index is the target value:
                        # Return the middle index
                # Else if the value at the middle index is less than our target value:
                        # Update pointer(s) to only search right half of the list in next loo
        # If we search whole list and haven't found target value, return -1

def binary_search(lst, target):
        pass
```

Example Usage:

```
# Example Input: lst = [1, 3, 5, 7, 9, 11, 13, 15], target = 11
```

Example Output:

```
# Expected Output: 5
# Explanation: 11 has index 5 in the list
```

▶ ✨ **AI Hint: Binary Search**

Problem 6: Backwards Binary Search

Generally binary search returns the index of the **first occurrence** of the `target` in the list. Write an updated version of binary search `find_last()` that, given a list that may contain duplicates, returns the index of the last occurrence of `target`.

Evaluate the time and space complexity of your function.

```
def find_last(lst, target):
        pass
```

Example Usage:

```
# Example Input: lst = [1, 3, 5, 7, 9, 11, 11, 13, 15], target = 11
```

Example Output:

```
# Expected Output: 6
# Explanation: The second (last) occurrence of 11 has index 6 in the list
```

Problem 7: Find Floor

Given a sorted list of integers and a value `x`, return the index of the floor of `x`. The floor of `x` is the largest element in the array smaller than or equal to `x`. If there is no floor of `x`, return `-1`.

Evaluate the time and space complexity of your function.

Example Usage:

```
# Example Input: lst = [1, 2, 8, 10, 11, 12, 19], x = 5
```

Example Output:

```
# Expected Output: 1
# 2 is the largest element in the list that is less than or equal to 5. 2 has index 1 in the
```

Close Section

▸ **Problem Set Version 2**

▸ **Problem Set Version 3**