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: Review I

Overview

Review of different topics covered in units 1-9. Be prepared to answer from a variety of different types of questions, invoking different algorithms and using different data structures.

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 10 Cheatsheet

This cheatsheet outlines content that will enhance and optimize your solutions to Unit 10 problems but *are not strictly necessary to solve the problems*.

⚠️ Approaching Unit 10 Problems

To help transition you from the CodePath classroom to an independent study environment post-classroom and to better prepare you for interviews, this unit does not contain hints.

Problems may cover any topic from Unit 1-9, and it is your job to match the problem to the most appropriate algorithmic technique and/or data structure. Some problems may not match any specific technique, but will require to use your general knowledge and the problem solving skills you have strengthened over the past nine units to find a solution.

Some problems may be repeated to you from past units. This is *intentional*. Use these problems as an opportunity to review techniques. For an extra challenge, use your additional knowledge to come to an even better solution than the one you developed earlier in the course. Check the cheatsheet for syntax and data structures that will help you write cleaner, more optimal code!

Happy coding!

▾ Problem Set Version 1

Problem 1: Valid Parentheses

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, return `True` if the input string is valid and `False` otherwise.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

```
def is_valid(s):
        pass
```

Example Usage:

```
Example #1:
Input: s = "()"
Expected Output: True

Example #2:
s = "()[]{}"
Expected Output: True

Example #3:
s = "(())"
Expected Output: True

Example #4:
s = "(]"
Expected Output: False

Example #5:
s = "([)]"
Expected Output: False
```

Problem 2: Best Time to Buy & Sell Stock

You are given a list of integers `prices` where `prices[i]` is the price of a given stock on the `ith` day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different**

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

```python
def max_profit(prices):
	pass
```

```
Example #1:
Input: prices = [7,1,5,3,6,4]
Expected Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you

Example #2:
Input: prices = [7,6,4,3,1]
Expected Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.
```

Problem 3: Shuffle Merge

Given the heads of two singly linked lists of integers, merge their nodes to make one list, taking nodes alternately between the two lists. If either list runs out of elements before the other, all nodes from the list with remaining nodes should be appended onto the end of the merged list. Return the head of the merged list.

```python
def shuffle_merge(head_a, head_b):
	pass
```

```

Input Lists: List 1: 1 -> 2 -> 3, List 2: 4 -> 5 -> 6
Input: head_a = 1, head_b = 4
Expected Return value: 1
Expected Result List: 1 -> 4 -> 2 -> 5 -> 3 -> 6

Input Lists: List 1: 1 -> 2 -> 3, List 2: 4
Expected Return value: 1
Expected Result List: 1 -> 4 -> 2 -> 3
```

Problem 4: Group Anagrams

Given an array of strings `strs`, group **the anagrams** together. You can return the answer in **any**

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

```
def group_anagrams(strs):
        pass
```

Example Usage:

```
Example #1:
Input: strs = ["eat","tea","tan","ate","nat","bat"]
Expeced Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Example #2:
Input: strs = [""]
Expected Output: [[""]]

Example #3:
Input: strs = ["a"]
Expected Output: [["a"]]
```

Problem 5: Sum Root to Leaf Numbers

You are given the `root` of a binary tree containing digits from `0` to `9` only.

Each root-to-leaf path in the tree represents a number.

- For example, the root-to-leaf path `1 -> 2 -> 3` represents the number `123`.

Return *the total sum of all root-to-leaf numbers*.

A **leaf** node is a node with no children.

```
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def sum_numbers(root):
        pass
```

Example Usage:

```
 Example Input Tree #1:

      1
     / \
Example Input: root = 1
Expected Output: 25
Explanation:
The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Therefore, sum = 12 + 13 = 25.

Example Input Tree #2:

      4
     / \
    9   0
   / \
  5   1

Input: root = 4
Expected Output: 1026
Explanation:
The root-to-leaf path 4->9->5 represents the number 495.
The root-to-leaf path 4->9->1 represents the number 491.
The root-to-leaf path 4->0 represents the number 40.
Therefore, sum = 495 + 491 + 40 = 1026.
```

Close Section

▸ Problem Set Version 2

▸ Problem Set Version 3