TIP102 | Intermediate Technical Interview Prep

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

Session 2: Strings & Arrays

Session Overview

Students will build upon the concepts introduced in the first session by tackling more complex string and array problems. They will deepen their understanding of manipulating data structures, including reversing words in a sentence, summing digits, and handling list operations such as finding exclusive elements between two lists. The session will also introduce them to important problem-solving techniques, like using loops and conditionals to manipulate numbers and strings.

> 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 2

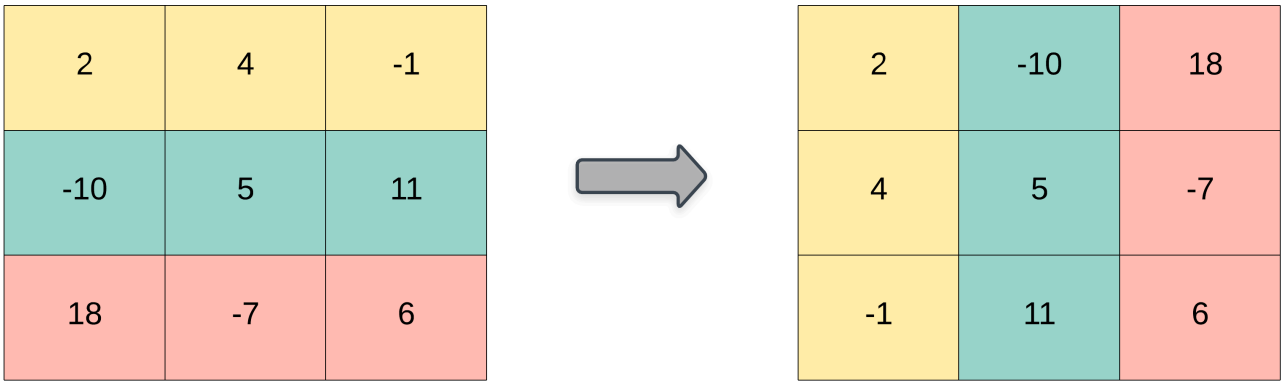
▶ **Standard Problem Set Version 1**

▶ **Standard Problem Set Version 2**

▼ **Advanced Problem Set Version 1**

Problem 1: Transpose Matrix

Write a function `transpose()` that accepts a 2D integer array `matrix` and returns the transpose of `matrix`. The transpose of a matrix is the matrix flipped over its main diagonal, swapping the rows and columns.




```
def transpose(matrix):
    pass
```

Example Usage

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
transpose(matrix)

matrix = [
    [1, 2, 3],
    [4, 5, 6]
]
transpose(matrix)
```

Example Output:

```
[
    [1, 4, 7],
    [2, 5, 8],
    [3, 6, 9]
]
[
    [1, 4],
    [2, 5],
    [3, 6]
]
```

▶ 💡 **Hint: Nested Lists**

▶ 💡 **Hint: Nested Loops**

Problem 2: Two-Pointer Reverse List

Write a function `reverse_list()` that takes in a list `lst` and returns elements of the list in reverse order. The list should be reversed in-place without using list slicing (e.g. `lst[::-1]`).

Instead, use the two-pointer approach, which is a common technique in which we initialize two variables (also called a pointer in this context) to track different indices or places in a list or string, then moves the pointers to point at new indices based on certain conditions. In the most common variation of the two-pointer approach, we initialize one variable to point at the beginning of a list and a second variable/pointer to point at the end of list. We then shift the pointers to move inwards through the list towards each other, until our problem is solved or the pointers reach the opposite ends of the list.

```
def reverse_list(lst):
    pass
```

Example Usage

```
lst = ["pooh", "christopher robin", "piglet", "roo", "eeyore"]
reverse_list(lst)
```

Example Output:

```
["eeyore", "roo", "piglet", "christopher robin", "pooh"]
```

▶ 💡 **Hint: While Loops**

Problem 3: Remove Duplicates

Write a function `remove_dupes()` that accepts a sorted array `items`, and removes the duplicates in-place such that each element appears only once. Return the length of the modified array. You may not create another array; your implementation must modify the original input array `items`.

```
def remove_dupes(items):
    pass
```

Example Usage

```
items = ["extract of malt", "haycorns", "honey", "thistle", "thistle"]
remove_dupes(items)

items = ["extract of malt", "haycorns", "honey", "thistle"]
remove_dupes(items)
```

Example Output:

```
4
4
```

▶ 💡 **Hint: Two Pointer Technique**

Problem 4: Sort Array by Parity

Given an integer array `nums`, write a function `sort_by_parity()` that moves all the even integers at the beginning of the array followed by all the odd integers.

Return ***any array*** *that satisfies this condition*.

```
def sort_by_parity(nums):
    pass
```

Example Usage

```
nums = [3, 1, 2, 4]
sort_by_parity(nums)

nums = [0]
sort_by_parity(nums)
```

Example Output:

```
[2, 4, 3, 1]
[0]
```

▶ 💡 **Remainders with Modulus Division**

Problem 5: Container with Most Honey

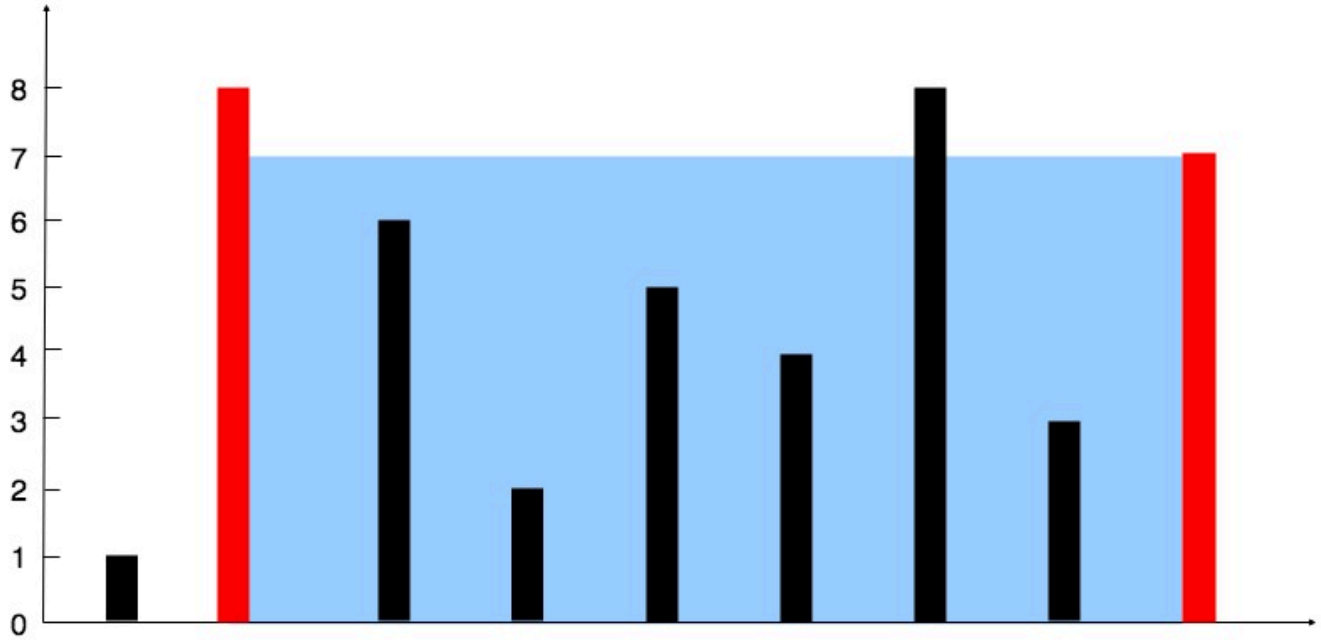
Christopher Robin is helping Pooh construct the biggest hunny jar possible. Help his write a function that accepts an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `ith` line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most honey.

Return the maximum amount of honey a container can store.

Notice that you may not slant the container.

```
def most_honey(height):
        pass
```



Example Usage

```
height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
most_honey(height)

height = [1, 1]
most_honey(height)
```

Example Output:

```
49
1
```

Problem 6: Merge Intervals

Write a function `merge_intervals()` that accepts an array of `intervals` where `intervals[i] = [starti, endi]`, merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

```
def merge_intervals(intervals):
        pass
```

Example Usage

```
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
merge_intervals(intervals)

intervals = [[1, 4], [4, 5]]
merge_intervals(intervals)
```

Example Output:

```
[[1, 6], [8, 10], [15, 18]]
[[1, 5]]
```

▶ 💡 **Hint: Sorting Lists**

Close Section

▼ Advanced Problem Set Version 2

Problem 1: Matrix Addition

Write a function `add_matrices()` that accepts to `n x m` matrices `matrix1` and `matrix2`. The function should return an `n x m` matrix `sum_matrix` that is the sum of the given matrices such that each value in `sum_matrix` is the sum of values of corresponding elements in `matrix1` and `matrix2`.

```
def add_matrices(matrix1, matrix2):
    pass
```

Example Usage

```
matrix1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

matrix2 = [
    [9, 8, 7],
    [6, 5, 4],
    [3, 2, 1]
]

add_matrices(matrix1, matrix2)
```

Example Output:

```
[
    [10, 10, 10],
    [10, 10, 10],
    [10, 10, 10]
]
```

▶ 💡 **Hint: Nested Lists**

▶ 💡 **Hint: Nested Loops**

Problem 2: Two-Pointer Palindrome

Write a function `is_palindrome()` that takes in a string `s` as a parameter and returns `True` if the string is a palindrome and `False` otherwise. You may assume the string contains only lowercase alphabetic characters.

The function must use the two-pointer approach, which is a common technique in which we initialize two variables (also called a pointer in this context) to track different indices or places in a list or string, then moves the pointers to point at new indices based on certain conditions. In the most common variation of the two-pointer approach, we initialize one variable to point at the beginning of a list and a second variable/pointer to point at the end of list. We then shift the pointers to move inwards through the list towards each other, until our problem is solved or the pointers reach the opposite ends of the list.

```
def is_palindrome(s):
    pass
```

Example Usage

```
s = "madam"
is_palindrome(s)

s = "madamweb"
is_palindrome(s)
```

Example Output:

```
True
False
```

▶ 💡 **Hint: While Loops**

▶ 💡 **Hint: Two Pointer Technique**

Problem 3: Squash Spaces

Write a function `squash_spaces()` that takes in a string `s` as a parameter and returns a new string with each substring with consecutive spaces reduced to a single space. Assume `s` can contain leading or trailing spaces, but in the result should be trimmed. Do not use any of the built-in trim methods.

```
def squash_spaces(s):
    pass
```

Example Usage

```
s = "   Up,     up,   and  away! "
squash_spaces(s)

s = "With great power comes great responsibility."
squash_spaces(s)
```

Example Output:

```
"Up, up, and away!"
"With great power comes great responsibility."
```

Problem 4: Two-Pointer Two Sum

Use the two pointer approach to implement a function `two_sum()` that takes in a sorted list of integers `nums` and an integer `target` as parameters and returns the indices of the two numbers that add up to `target`. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the indices in any order.

```
def two_sum(nums, target):
    pass
```

Example Usage

```
nums = [2, 7, 11, 15]
target = 9
two_sum(nums, target)

nums = [2, 7, 11, 15]
target = 18
two_sum(nums, target)
```

Example Output:

```
[0, 1]
[1, 2]
```

Problem 5: Three Sum

Given an integer array `nums`, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

```
def three_sum(nums):
    pass
```

Example Usage

```
nums = [-1, 0, 1, 2, -1, -4]
three_sum(nums)

nums = [0, 1, 1]
three_sum(nums)

nums = [0, 0, 0]
three_sum(nums)
```

Example Output:

```
[[-1, -1, 2], [-1, 0, 1]]
[]
[[0, 0, 0]]
```

▶ 💡 **Hint: Sorting Lists**

Problem 6: Insert Interval

Implement a function `insert_interval()` that accepts an array of non-overlapping intervals `intervals` where `intervals[i] = [starti, endi]` represent the start and the end of the `ith` interval and `intervals` is sorted in ascending order by `starti`. The function also accepts an interval `new_interval = [start, end]` that represents the start and end of another interval.

Insert `new_interval` into `intervals` such that `intervals` is still sorted in ascending order by `starti` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` after the insertion.

You don't need to modify intervals in-place. You can make a new array and return it.

```
def insert_interval(intervals, new_interval):
    pass
```

Example Usage

```
intervals = [[1, 3], [6, 9]]
new_interval = [2, 5]
insert_interval(intervals, new_interval)

intervals = [[1, 2], [3, 5], [6, 7], [8, 10], [12, 16]]
new_interval = [4, 8]
insert_interval(intervals, new_interval)
```

Example Output:

```
[[1, 5], [6, 9]]
[[1, 2], [3, 10], [12, 16]]
```

Close Section