TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: 126663

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

While **each problem may call for slightly different approaches to these three steps**, the basics of the steps won't change, and it's important to engage them each time. We've built out some starting points to use in our breakout sessions, below!

Please read the following carefully (take 10 minutes as a team, if you like) and follow these basic steps, as a group, through each of the problems in your problem set.

Fun Fact: We sometimes call the main beats of problem solving, or the tasks that teams are being asked to take, our "DoNow's". If you hear a staff member using this phrase, (e.g., "Ok great! Your team is struggling with the Understand step – what might be a DoNow?") you now know what they might mean!

▼ UPI Example

Step	What is it?	Try It!
1. Understand	Here we strive to **Understand** what the interviewer is asking for. It's common to restate the problem aloud, ask questions, and consider related test cases.	• Nominate one person to share their screen so everyone is on the same page, and bring up the problem at hand. (NOTE: Please trade-off and change who is screen sharing, roughly each problem • Have one person read the problem aloud. • Have a different person restate the problem in their own words. • Have members of the group ask 2-3 questions about the problem, perhaps about the example usage, or expected output • Example: "Will the list always contain only numbers?"

2. Plan	Then we **Plan** a solution, starting with appropriate visualizations and pseudocode.	• Restate - have one person share the general idea about what the function is trying to accomplish.
		• Next, break down the problem into subproblems as a group. Each member should participate.
		  ○ If you don't know where to start, try to describe how you would solve the problem *without a computer*.
		• As a group, translate each subproblem into pseudocode.
		  ○ How do I do what I described in English in Python?
		  ○ Then, do I need to change my approach to any steps to make it work in code?

3. Implement	Now we **Implement** our solution, by translating our **Plan** into Python.	• Translate the pseudocode into Python—this is a stage at which you can consider working individually.

Breakout Problems Session 2

▾ Standard Problem Set Version 1

Problem 1: Reverse Sentence

Write a function `reverse_sentence()` that takes in a string `sentence` and returns the sentence with the order of the words reversed. The sentence will contain only alphabetic characters and spaces to separate the words. If there is only one word in the sentence, the function should return the original string.

```
def reverse_sentence(sentence):
    pass
```

Example Usage:

```
sentence = "tubby little cubby all stuffed with fluff"
reverse_sentence(sentence)

sentence = "Pooh"
reverse_sentence(sentence)
```

Example Output:

```
"fluff with stuffed all cubby little tubby"
"Pooh"
```

▼ 💡 **Hint: String Methods**

When working with strings, it's very common to need to process the string to convert all characters to upper or lower case, remove punctuation, handle whitespace, etc. Luckily, Python has several built-in string methods for common string operations. Practice your research skills by looking up common string methods to find one that will help you implement this function, or check out the Unit 1 cheatsheet for the most essential ones.

Problem 2: Goldilocks Number

In the extended universe of fictional bears, Goldilocks finds an enticing list of numbers in the Three Bears' house. She doesn't want to take a number that's too high or too low - she wants a number that's juuust right. Write a function `goldilocks_approved()` that takes in the list of distinct positive integers `nums` and returns any number from the list that is neither the minimum nor the maximum value in the array, or `-1` if there is no such number.

Return the selected integer.

```
def goldilocks_approved(nums):
    pass
```

Example Usage

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

nums = [1, 2]
goldilocks_approved(nums)

nums = [2, 1, 3]
goldilocks_approved(nums)
```

Example Output:

```
2
-1
2
```

▼ 💡 **Hint: Minimums and Maximums**

This problem may benefit from knowing how to use Python's `min()` or `max()` functions to find the minimum or maximum of a sequence of values. Use your independent research skills or the unit cheatsheet to read more about how to use these functions!

Problem 3: Delete Minimum

Pooh is eating all of his hunny jars in order of smallest to largest. Given a list of integers `hunny_jar_sizes`, write a function `delete_minimum_elements()` that continuously removes the minimum element until the list is empty. Return a new list of the elements of `hunny_jar_sizes` in the order in which they were removed.

```
def delete_minimum_elements(hunny_jar_sizes):
        pass
```

Example Usage

```
hunny_jar_sizes = [5, 3, 2, 4, 1]
delete_minimum_elements(hunny_jar_sizes)

hunny_jar_sizes = [5, 2, 1, 8, 2]
delete_minimum_elements(hunny_jar_sizes)
```

Example Output:

```
[1, 2, 3, 4, 5]
[1, 2, 2, 5, 8]
```

▼ 💡 **Hint: While Loops**

This problem may benefit from a while loop! If you are unfamiliar with while loop syntax in Python, use your independent research skills or the Python Syntax section of the unit cheatsheet to learn more.

Problem 4: Sum of Digits

Write a function `sum_of_digits()` that accepts an integer `num` and returns the sum of `num`'s digits.

```
def sum_of_digits(num):
    pass
```

Example Usage

```
num = 423
sum_of_digits(num)

num = 4
sum_of_digits(num)
```

Example Output:

```
9 # Explanation: 4 + 2 + 3 = 9
4
```

▼ 💡 **Hint: Floor Division**

This problem may benefit from either floor division, which is where the result of dividing two numbers is rounded down. Use a search engine or a generative AI tool to research how to perform floor division in Python.

Problem 5: Bouncy, Flouncy, Trouncy, Pouncy

Tigger has developed a new programming language Tiger with only **four** operations and **one** variable `tigger`.

- `bouncy` and `flouncy` both **increment** the value of the variable `tigger` by `1`.
- `trouncy` and `pouncy` both **decrement** the value of the variable `tigger` by 1.

Initially, the value of `tigger` is `1` because he's the only tigger around! Given a list of strings `operations` containing a list of operations, return the **final** value of `tigger` after performing all the operations.

```
def final_value_after_operations(operations):
        pass
```

Example Usage

```
operations = ["trouncy", "flouncy", "flouncy"]
final_value_after_operations(operations)

operations = ["bouncy", "bouncy", "flouncy"]
final_value_after_operations(operations)
```

Example Output:

```
2
4
```

Problem 6: Acronym

Given an array of strings `words` and a string `s`, implement a function `is_acronym()` that returns `True` if `s` is an acronym of `words` and returns `False` otherwise.

The string `s` is considered an acronym of `words` if it can be formed by concatenating the first character of each string in `words` in order. For example, `"pb"` can be formed from `["pooh"", "bear"]`, but it can't be formed from `["bear", "pooh"]`.

```
def is_acronym(words, s):
    pass
```

Example Usage

```
words = ["christopher", "robin", "milne"]
s = "crm"
is_acronym(words, s)
```

Example Output:

```
True
```

Problem 7: Good Things Come in Threes

Write a function `make_divisible_by_3()` that accepts an integer array `nums`. In one operation, you can add or subtract `1` from any element of `nums`. Return the minimum number of operations to make all elements of nums divisible by 3.

```
def make_divisible_by_3(nums):
    pass
```

Example Usage

```
nums = [1, 2, 3, 4]
make_divisible_by_3(nums)

nums = [3, 6, 9]
make_divisible_by_3(nums)
```

Example Output:

```
3
0
```

▼ 💡 **Remainders with Modulus Division**

This problem requires you to know how to find the remainder of a division operation. We can do this with something called modulus division. If you are unfamiliar with how to do this in Python, checkout the Unit 1 cheatsheet or do your own research.

Problem 8: Exclusive Elements

Given two lists `lst1` and `lst2`, write a function `exclusive_elemts()` that returns a new list that contains the elements which are in `lst1` but not in `lst2` and the elements that are in `lst2` but not in `lst1`.

```
def exclusive_elemts(lst1, lst2):
        pass
```

Example Usage

```
lst1 = ["pooh", "roo", "piglet"]
lst2 = ["piglet", "eeyore", "owl"]
exclusive_elemts(lst1, lst2)

lst1 = ["pooh", "roo"]
lst2 = ["piglet", "eeyore", "owl", "kanga"]
exclusive_elemts(lst1, lst2)

lst1 = ["pooh", "roo", "piglet"]
lst2 = ["pooh", "roo", "piglet"]
exclusive_elemts(lst1, lst2)
```

Example Output:

```
["pooh", "roo", "eeyore", "owl"]
["pooh", "roo", "piglet", "eeyore", "owl", "kanga"]
[]
```

Problem 9: Merge Strings Alternately

Write a function `merge_alternately()` that accepts two strings `word1` and `word2`. Merge the strings by adding letters in alternating order, starting with `word1`. If a string is longer than the other, append the additional letters onto the end of the merged string.

Return the merged string.

```
def merge_alternately(word1, word2):
    pass
```

Example Usage

```
word1 = "wol"
word2 = "oze"
merge_alternately(word1, word2)

word1 = "hfa"
word2 = "eflump"
merge_alternately(word1, word2)

word1 = "eyre"
word2 = "eo"
merge_alternately(word1, word2)
```

Example Output:

```
"woozle"
"heffalump"
"eeyore"
```

Problem 10: Eeyore's House

Eeyore has collected two piles of sticks to rebuild his house and needs to choose pairs of sticks whose lengths are the right proportion. Write a function `good_pairs()` that accepts two integer arrays `pile1` and `pile2` where each integer represents the length of a stick. The function also accepts a positive integer `k`. The function should return the number of **good** pairs.

A pair `(i, j)` is called **good** if `pile1[i]` is divisible by `pile2[j] * k`. Assume `0 <= i <= len(pile1) - 1` and `0 <= j <= len(pile2) - 1`

```
def good_pairs(pile1, pile2, k):
        pass
```

Example Usage

```
pile1 = [1, 3, 4]
pile2 = [1, 3, 4]
k = 1
good_pairs(pile1, pile2, k)

pile1 = [1, 2, 4, 12]
pile2 = [2, 4]
k = 3
good_pairs(pile1, pile2, k)
```

Example Output:

```
5
2
```

▼ ✨ AI Hint: Nested Loops

Key Skill: Use AI to explain code concepts

This problem may benefit from an understanding of nested loops. For a refresher, check out the Advanced section of the Unit 1 Cheatsheet.

Want to dive deeper? Ask an AI tool like ChatGPT or GitHub Copilot to show you examples of how to work with nested loops in Python.

Close Section

▼ Standard Problem Set Version 2

Problem 1: String Array Equivalency

Given two string arrays `word1` and `word2`, return `True` if the two arrays **represent** the same string, and `False` otherwise.

A string is **represented** by an array if the array elements concatenated in order forms the string.

```
def are_equivalent(word1, word2):
    pass
```

Example Usage

```
word1 = ["bat", "man"]
word2 = ["b", "atman"]
are_equivalent(word1, word2)

word1 = ["alfred", "pennyworth"]
word2 = ["alfredpenny", "word"]
are_equivalent(word1, word2)

word1  = ["cat", "wom", "an"]
word2 = ["catwoman"]
are_equivalent(word1, word2)
```

Example Output:

```
True
False
True
```

▼ 💡 **Hint: String Methods**

When working with strings, it's very common to need to process the string to convert all characters to upper or lower case, remove punctuation, handle whitespace, etc. Luckily, Python has several built-in string methods for common string operations. Practice your research skills by looking up common string methods to find one that will help you implement this function, or check out the Unit 1 cheatsheet for the most essential ones.

Problem 2: Count Even Strings

Implement a function `count_evens()` that accepts a list of strings `lst` as a parameter. The function should return the number of strings with an even length in the list.

```
def count_evens(lst):
    pass
```

Example Usage

```
lst = ["na", "nana", "nanana", "batman", "!"]
count_evens(lst)

lst = ["the", "joker", "robin"]
count_evens(lst)

lst = ["you", "either", "die", "a", "hero", "or", "you", "live", "long", "enough", "to", "se
count_evens(lst)
```

Example Output:

```
4
0
9
```

▼ 💡 **Remainders with Modulus Division**

This problem requires you to know how to find the remainder of a division operation. We can do this with something called modulus division. If you are unfamiliar with how to do this in Python, checkout the Unit 1 cheatsheet or do your own research.

Problem 3: Secret Identity

Write a function `remove_name()` to keep Batman's secret identity hidden. The function accepts a list of names `people` and a string `secret_identity` and should return the list with any instances of `secret_identity` removed. The list must be modified in place; you may not create any new lists as part of your solution. Relative order of the remaining elements must be maintained.

```
def remove_name(people, secret_identity):
        pass
```

Example Usage

```
people = ['Batman', 'Superman', 'Bruce Wayne', 'The Riddler', 'Bruce Wayne']
secret_identity = 'Bruce Wayne'
remove_name(people, secret_identity)
```

Example Output:

```
['Batman', 'Superman', 'The Riddler']
```

▼ 💡 **Hint: While Loops**

This problem may benefit from a while loop! If you are unfamiliar with while loop syntax in Python, use your independent research skills or the Python Syntax section of the unit cheatsheet to learn more.

Problem 4: Count Digits

Given a non-negative integer `n`, write a function `count_digits()` that returns the number of digits in `n`. You may not cast `n` to a string.

```
def count_digits(n):
    pass
```

Example Usage

```
n = 964
count_digits(n)

n = 0
count_digits(n)
```

Example Output:

```
3
1
```

▼ 💡 **Hint: Floor Division**

This problem may benefit from either floor division, which is where the result of dividing two numbers is rounded down. Use a search engine or a generative AI tool to research how to perform floor division in Python.

Problem 5: Move Zeroes

Write a function `move_zeroes` that accepts an integer array `nums` and returns a new list with all `0`s moved to the end of list. The relative order of the non-zero elements in the original list should be maintained.

```
def move_zeroes(lst):
    pass
```

Example Usage

```
lst = [1, 0, 2, 0, 3, 0]
move_zeroes(lst)
```

Example Output:

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

Problem 6: Reverse Vowels of a String

Given a string `s`, reverse only all the vowels in the string and return it.

The vowels are `'a'`, `'e'`, `'i'`, `'o'`, and `'u'`, and they can appear in both lower and upper cases and more than once.

```
def reverse_vowels(s):
    pass
```

Example Usage

```
s = "robin"
reverse_vowels(s)

s = "BATgirl"
reverse_vowels(s)

s = "batman"
reverse_vowels(s)
```

Example Output:

```
"ribon"
"BiTgArl"
"batman"
```

Problem 7: Vantage Point

Batman is going on a scouting trip, surveying an area where he thinks Harley Quinn might commit her next crime spree. The area has many hills with different heights and Batman wants to find the tallest one to get the best vantage point. His scout trip consists of `n + 1` points at different altitudes. Batman starts his trip at point `0` with altitude `0`.

Write a function `highest_altitude()` that accepts an integer array `gain` of length `n` where `gain[i]` is the net gain in altitude between points `i` and `i + 1` for all `(0 <= i < n)`. Return the highest altitude of a point.

```
def highest_altitude(gain):
    pass
```

Example Usage

```
gain = [-5, 1, 5, 0, -7]
highest_altitude(gain)

gain = [-4, -3, -2, -1, 4, 3, 2]
highest_altitude(gain)
```

Example Output:

```
1
0
```

Problem 8: Left and Right Sum Differences

Given a 0-indexed integer array `nums`, write a function `left_right_difference` that returns a 0-indexed integer array `answer` where:

- `len(answer) == len(nums)`
- `answer[i]` = `left_sum[i] - right_sum[i]`

Where:

- `left_sum[i]` is the sum of elements to the left of the index `i` in the array `nums`. If there is no such element, `left_sum[i] = 0`
- `right_sum[i]` is the sum of elements to the right of the index `i` in the array `nums`. If there is no such element, `right_sum[i] = 0`

```
def left_right_difference(nums):
    pass
```

Example Usage

```
nums = [10, 4, 8, 3]
left_right_difference(nums)

nums = [1]
left_right_difference(nums)
```

Example Output:

```
[-15, -1, 11, 22]
[0]
```

Problem 9: Common Cause

Write a function `common_elements()` that takes in two lists `lst1` and `lst2` and returns a list of the elements that are common to both lists.

```
def common_elements(lst1, lst2):
    pass
```

```
Example 1:
Input: lst1 = ["super strength", "super speed", "x-ray vision"], lst2 = ["super speed", "time
Output: ["super speed"]

Example 2:
Input: lst1 = ["super strength", "super speed", "x-ray vision"], lst2 = ["martial arts", "ste
Output: []
```

Example Usage

```
lst1 = ["super strength", "super speed", "x-ray vision"]
lst2 = ["super speed", "time travel", "dimensional travel"]
common_elements(lst1, lst2)

lst1 = ["super strength", "super speed", "x-ray vision"]
lst2 = ["martial arts", "stealth", "master detective"]
common_elements(lst1, lst2)
```

Example Output:

```
["super speed"]
[]
```

▼ ✨ AI Hint: Nested Loops

Key Skill: Use AI to explain code concepts

This problem may benefit from an understanding of nested loops. For a refresher, check out the Advanced section of the Unit 1 Cheatsheet.

Want to dive deeper? Ask an AI tool like ChatGPT or GitHub Copilot to show you examples of how to work with nested loops in Python.

Problem 10: Exposing Superman

Metropolis has a population `n`, with each citizen assigned an integer id from `1` to `n`. There's a rumor that Superman is an ordinary citizen among this group.

If Superman is an ordinary citizen, then:

- Superman trusts nobody.
- Everybody (except for Superman) trusts Superman.
- There is exactly one citizen that satisfies properties 1 and 2.

Write a function `expose_superman()` that accepts a 2D array `trust` where `trust[i] = [ai, bi]` representing that the person labeled `ai` trusts the person labeled `bi`. If a trust relationship does not exist in `trust` array, then such a trust relationship does not exist.

Return the label of Superman if he is hiding amongst the population and can be identified, or return `-1` otherwise.

```
def expose_superman(trust, n):
    pass
```

Example Usage

```
n = 2
trust = [[1, 2]]
expose_superman(trust, n)

n = 3
trust = [[1, 3], [2, 3]]
expose_superman(trust, n)

n = 3
trust = [[1, 3], [2, 3], [3, 1]]
expose_superman(trust, n)
```

Example Output:

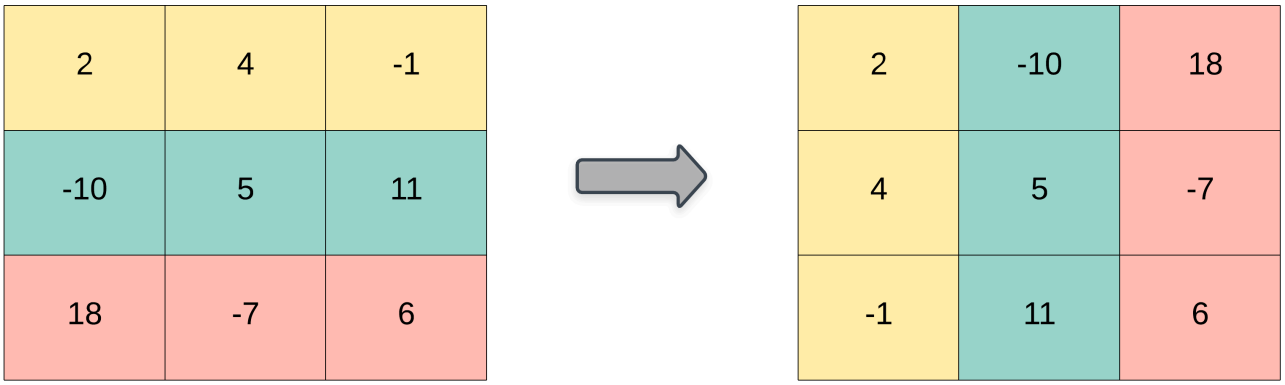
```
2
3
-1
```

Close Section

▾ 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]
]
```

▼ ✨ AI Hint: Nested Lists

Key Skill: Use AI to explain code concepts

This problem may benefit from an understanding of nested data, particularly nested lists. For a refresher, check out the Advanced section of the Unit 1 Cheatsheet.

Want to dive deeper? Ask an AI tool like ChatGPT or GitHub Copilot to show you examples of how to work with nested lists in Python.

▼ ✨ AI Hint: Nested Loops

Key Skill: Use AI to explain code concepts

This problem may benefit from an understanding of nested loops. For a refresher, check out the Advanced section of the Unit 1 Cheatsheet.

Want to dive deeper? Ask an AI tool like ChatGPT or GitHub Copilot to show you examples of how to work with nested loops in Python.

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**

This problem may benefit from a while loop! If you are unfamiliar with while loop syntax in Python, use your independent research skills or the Python Syntax section of the unit cheatsheet to learn more.

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**

This problem may benefit from a algorithmic technique called the **two pointer approach**. Take a look at the unit cheatsheet for a more in-depth overview of how this technique works.

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**

This problem requires you to know how to find the remainder of a division operation. We can do this with something called modulus division. If you are unfamiliar with how to do this in Python, checkout the Unit 1 cheatsheet or do your own research.

Problem 5: Container with Most Honey

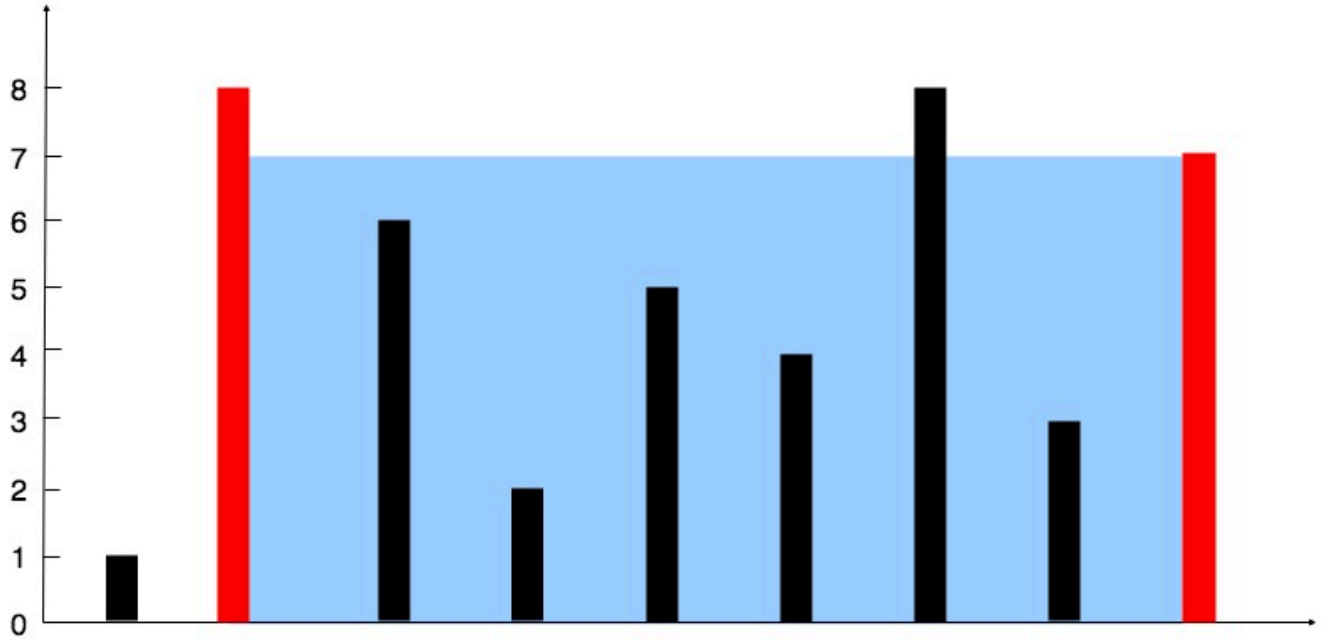
Christopher Robin is helping Pooh construct the biggest hunny jar possible. Help his write a function that accepts an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `ith` line are `(i, 0)` and `(i, height[i])`.

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**

This problem may benefit from knowing how to sort a list. Python provides a couple options for sorting lists and other iterables, including `sort()` and `sorted()`. Use your independent research skills or the unit cheatsheet to research how these functions work!

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]
]
```

▼ ✨ AI Hint: Nested Lists

Key Skill: Use AI to explain code concepts

This problem may benefit from an understanding of nested data, particularly nested lists. For a refresher, check out the Advanced section of the Unit 1 Cheatsheet.

Want to dive deeper? Ask an AI tool like ChatGPT or GitHub Copilot to show you examples of how to work with nested lists in Python.

▼ ✨ AI Hint: Nested Loops

Key Skill: Use AI to explain code concepts

This problem may benefit from an understanding of nested loops. For a refresher, check out the Advanced section of the Unit 1 Cheatsheet.

Want to dive deeper? Ask an AI tool like ChatGPT or GitHub Copilot to show you examples of how to work with nested loops in Python.

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**

This problem may benefit from a while loop! If you are unfamiliar with while loop syntax in Python, use your independent research skills or the Python Syntax section of the unit cheatsheet to learn more.

▼ 💡 **Hint: Two Pointer Technique**

This problem may benefit from a algorithmic technique called the **two pointer approach**. Take a look at the unit cheatsheet for a more in-depth overview of how this technique works.

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**

This problem may benefit from knowing how to sort a list. Python provides a couple options for sorting lists and other iterables, including `sort()` and `sorted()`. Use your independent research skills or the unit cheatsheet to research how these functions work!

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