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: Stacks, Queues, and Two Pointer

Session Overview

In this session we will continue to work with linear data structures like strings and arrays. Students will strengthen their ability to solve problems using stacks, queues, and the two pointer method.

> 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: Manage Performance Stage Changes

At a cultural festival, multiple performances are scheduled on a single stage. However, due to last-minute changes, some performances need to be rescheduled or canceled. The festival organizers use a stack to manage these changes efficiently.

You are given a list `changes` of strings where each string represents a change action. The actions can be:

- `"Schedule X"` : Schedule a performance with ID X on the stage.
- `"Cancel"` : Cancel the most recently scheduled performance that hasn't been canceled yet.
- `"Reschedule"` : Reschedule the most recently canceled performance to be the next on stage.

Return a list of performance IDs that remain scheduled on the stage after all changes have been applied.

```
def manage_stage_changes(changes):
    pass
```

Example Usage:

```
print(manage_stage_changes(["Schedule A", "Schedule B", "Cancel", "Schedule C", "Reschedule"
print(manage_stage_changes(["Schedule A", "Cancel", "Schedule B", "Cancel", "Reschedule", "C
print(manage_stage_changes(["Schedule X", "Schedule Y", "Cancel", "Cancel", "Schedule Z"]))
```

Example Output:

```
["A", "C", "B", "D"]
[]
["Z"]
```

::ai

▼ ✨ AI Hint: Stacks

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **stack**, a data structure that follows the Last In, First Out (LIFO) principle.

If you are unfamiliar with stacks, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a stack is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a stack different from a list or a queue? Can you show me examples of each?"

:::

Problem 2: Queue of Performance Requests

You are organizing a festival and want to manage the queue of requests to perform. Each request has a priority. Use a queue to process the performance requests in the order they arrive but ensure that requests with higher priorities are processed before those with lower priorities. Return the order in which performances are processed.

```
def process_performance_requests(requests):
    pass
```

Example Usage:

```
print(process_performance_requests([(3, 'Dance'), (5, 'Music'), (1, 'Drama')]))
print(process_performance_requests([(2, 'Poetry'), (1, 'Magic Show'), (4, 'Concert'), (3, 'S
print(process_performance_requests([(1, 'Art Exhibition'), (3, 'Film Screening'), (2, 'Works
```

Example Output:

```
['Music', 'Dance', 'Drama']
['Concert', 'Stand-up Comedy', 'Poetry', 'Magic Show']
['Keynote Speech', 'Panel Discussion', 'Film Screening', 'Workshop', 'Art Exhibition']
```

::ai

▼ ✨ AI Hint: Queues

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **queue**, a data structure that follows the First In, First Out (FIFO) principle.

If you are unfamiliar with queues, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a queue is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a queue different from a list or stack? Can you show me examples of each?"

:::

Problem 3: Collecting Points at Festival Booths

At the festival, there are various booths where visitors can collect points. Each booth has a specific number of points available. Use a stack to simulate the process of collecting points and return the total points collected after visiting all booths.

```
def collect_festival_points(points):
    pass
```

Example Usage:

```
print(collect_festival_points([5, 8, 3, 10]))
print(collect_festival_points([2, 7, 4, 6]))
print(collect_festival_points([1, 5, 9, 2, 8]))
```

Example Output:

```
26
19
25
```

Problem 4: Festival Booth Navigation

At the cultural festival, you are managing a treasure hunt where participants need to visit booths in a specific order. The order in which they should visit the booths is defined by a series of clues. However, some clues lead to dead ends, and participants must backtrack to previous booths to continue their journey.

You are given a list of clues, where each clue is either a booth number (an integer) to visit or the word "back" indicating that the participant should backtrack to the previous booth.

Write a function to simulate the participant's journey and return the final sequence of booths visited, in the order they were visited.

```
def booth_navigation(clues):
    pass
```

Example Usage:

```
clues = [1, 2, "back", 3, 4]
print(booth_navigation(clues))

clues = [5, 3, 2, "back", "back", 7]
print(booth_navigation(clues))

clues = [1, "back", 2, "back", "back", 3]
print(booth_navigation(clues))
```

Example Output:

```
[1, 3, 4]
[5, 7]
[3]
```

Problem 5: Merge Performance Schedules

You are organizing a cultural festival and have two performance schedules, schedule1 and schedule2, each represented by a string where each character corresponds to a performance slot. Merge the schedules by adding performances in alternating order, starting with schedule1. If one schedule is longer than the other, append the additional performances onto the end of the merged schedule.

Return the merged performance schedule.

```
def merge_schedules(schedule1, schedule2):
    pass
```

Example Usage:

```
print(merge_schedules("abc", "pqr"))
print(merge_schedules("ab", "pqrs"))
print(merge_schedules("abcd", "pq"))
```

Example Output:

```
apbqcr
apbqrs
apbqcd
```

▼ 💡 **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 6: Next Greater Event

At a cultural festival, you have a schedule of events where each event has a unique popularity score. The schedule is represented by two distinct 0-indexed integer arrays `schedule1` and `schedule2`, where `schedule1` is a subset of `schedule2`.

For each event in `schedule1`, find its position in `schedule2` and determine the next event in `schedule2` with a higher popularity score. If there is no such event, then the answer for that event is `-1`.

Return an array `ans` of length `schedule1.length` such that `ans[i]` is the next greater event's popularity score as described above.

```
def next_greater_event(schedule1, schedule2):
  pass
```

Example Usage:

```
print(next_greater_event([4, 1, 2], [1, 3, 4, 2]))
print(next_greater_event([2, 4], [1, 2, 3, 4]))
```

Example Output:

```
[-1, 3, -1]
[3, -1]
```

Problem 7: Sort Performances by Type

You are organizing a cultural festival and have a list of performances represented by an integer array `performances`. Each performance is classified as either an even type (e.g., dance, music) or an odd type (e.g., drama, poetry).

Your task is to rearrange the performances so that all the even-type performances appear at the beginning of the array, followed by all the odd-type performances.

Return any array that satisfies this condition.

```
def sort_performances_by_type(performances):
  pass
```

Example Usage:

```
print(sort_performances_by_type([3, 1, 2, 4]))
print(sort_performances_by_type([0]))
```

Example Output:

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

Close Section

▼ Standard Problem Set Version 2

Problem 1: Final Costs After a Supply Discount

You are managing the budget for a global expedition, where the cost of supplies is represented by an integer array `costs`, where `costs[i]` is the cost of the `i` th supply item.

There is a special discount available during the expedition. If you purchase the `i` th item, you will receive a discount equivalent to `costs[j]`, where `j` is the minimum index such that `j > i` and `costs[j] <= costs[i]`. If no such `j` exists, you will not receive any discount.

Return an integer array `final_costs` where `final_costs[i]` is the final cost you will pay for the `i` th supply item, considering the special discount.

```
def final_supply_costs(costs):
  pass
```

Example Usage:

```
print(final_supply_costs([8, 4, 6, 2, 3]))
print(final_supply_costs([1, 2, 3, 4, 5]))
print(final_supply_costs([10, 1, 1, 6]))
```

Example Output:

```
[4, 2, 4, 2, 3]
[1, 2, 3, 4, 5]
[9, 0, 1, 6]
```

::ai

▼ ✨ AI Hint: Stacks

Key Skill: Use AI to explain code concepts

This problem may benefit from the use of a **stack**, a data structure that follows the Last In, First Out (LIFO) principle.

If you are unfamiliar with stacks, you can check the Unit 3 Cheatsheet for a refresher, or you can learn about them using a generative AI tool, like this:

"You're an expert computer science tutor. Please explain what a stack is in Python, and provide a simple code example of how to create one."

After you get your answer, you can also ask follow up questions:

"How is a stack different from a list or a queue? Can you show me examples of each?"

:::

Problem 2: Find First Symmetrical Landmark Name

During your global expedition, you encounter a series of landmarks, each represented by a string in the array `landmarks`. Your task is to find and return the first symmetrical landmark name. If there is no such name, return an empty string `""`.

A landmark name is considered symmetrical if it reads the same forward and backward.

```
def first_symmetrical_landmark(landmarks):
  pass
```

Example Usage:

```
print(first_symmetrical_landmark(["canyon","forest","rotor","mountain"]))
print(first_symmetrical_landmark(["plateau","valley","cliff"]))
```

Example Output:

```
rotor
```

▼ 💡 **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.

▼ 💡 **Hint: Helper Functions**

This problem may benefit from a helper function! If you find your functions getting too long or performing lots of different tasks, it might be a good indicator that you should add a helper function. Helper functions are functions we write to implement a subtask of our primary task. To learn more about helper functions (and inner functions, which is a common way to implement helper functions in Python), check out the unit cheatsheet!

Problem 3: Terrain Elevation Match

During your global expedition, you are mapping out the terrain elevations, where the elevation of each point is represented by an integer. You are given a string `terrain` of length `n`, where:

- `terrain[i] == 'I'` indicates that the elevation at the `i`th point is lower than the elevation at the `i`+1th point (`elevation[i] < elevation[i + 1]`).
- `terrain[i] == 'D'` indicates that the elevation at the `i`th point is higher than the elevation at the `i`+1th point (`elevation[i] > elevation[i + 1]`).

Your task is to reconstruct the elevation sequence and return it as a list of integers. If there are multiple valid sequences, return any of them.

Hint: Try using two variables: one to track the smallest available number and one for the largest. As you process each character in the string, assign the smallest number when the next elevation should increase ('I'), and assign the largest number when the next elevation should decrease ('D').

```
def terrain_elevation_match(terrain):
  pass
```

Example Usage:

```
print(terrain_elevation_match("IDID"))
print(terrain_elevation_match("III"))
print(terrain_elevation_match("DDI"))
```

Example Output:

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

Problem 4: Find the Expedition Log Concatenation Value

You are recording journal entries during a global expedition, where each entry is represented by a 0-indexed integer array, `logs`. The concatenation of two journal entries means combining their numerals into one.

For example, concatenating the numbers 15 and 49 results in 1549.

Your task is to calculate the total concatenation value of all the journal entries, which starts at 0. To do this, perform the following steps until no entries remain:

1. If there are at least two entries in the `logs`, concatenate the first and last entries, add the result to the current concatenation value, and then remove these two entries.
2. If there is only one entry left, add its value to the concatenation value and remove it from the array.
3.

Return the final concatenation value after all entries have been processed.

```
def find_the_log_conc_val(logs):
  pass
```

Example Usage:

```
print(find_the_log_conc_val([7, 52, 2, 4]))
print(find_the_log_conc_val([5, 14, 13, 8, 12]))
```

Example Output:

```
596
673
```

Problem 5: Number of Explorers Unable to Gather Supplies

During a global expedition, explorers must gather supplies from a limited stockpile, which includes two types of resources: type 0 (e.g., food rations) and type 1 (e.g., medical kits). The explorers are lined up in a queue, each with a specific preference for one of the two types of resources.

The number of supplies in the stockpile is equal to the number of explorers. The supplies are stacked in a pile. At each step:

- If the explorer at the front of the queue prefers the resource on the top of the stack, they will take it and leave the queue.

- Otherwise, they will leave the resource and go to the end of the queue.

This process continues until no explorer in the queue wants to take the top resource, leaving some explorers unable to gather the supplies they need.

You are given two integer arrays explorers and supplies, where `supplies[i]` is the type of the `i` th resource in the stack (`i = 0` is the top of the stack) and `explorers[j]` is the preference of the `j` th explorer in the initial queue (`j = 0` is the front of the queue). Return the number of explorers that are unable to gather their preferred supplies.

```
def count_explorers(explorers, supplies):
  pass
```

Example Usage:

```
print(count_explorers([1, 1, 0, 0], [0, 1, 0, 1]))
print(count_explorers([1, 1, 1, 0, 0, 1], [1, 0, 0, 0, 1, 1]))
```

Example Output:

```
0
3
```

Problem 6: Count Balanced Terrain Subsections

During your global expedition, you are analyzing a binary terrain string, `terrain`, where `0` represents a valley and `1` represents a hill. You need to count the number of non-empty *balanced subsections* in the terrain. A balanced subsection is defined as a contiguous segment of the terrain where an equal number of valleys (`0` s) and hills (`1` s) appear, and all the `0` s and `1` s are grouped consecutively.

Your task is to return the total number of these balanced subsections. Note that subsections that occur multiple times should be counted each time they appear.

```
def count_balanced_terrain_subsections(terrain):
    pass
```

Example Usage:

```
print(count_balanced_terrain_subsections("00110011"))
print(count_balanced_terrain_subsections("10101"))
```

Example Output:

```
6
4
```

Problem 7: Check if a Signal Occurs as a Prefix in Any Transmission

During your global expedition, you are monitoring various transmissions, each consisting of some signals separated by a single space. You are given a `searchSignal` and need to check if it occurs as a prefix to any signal in a transmission.

Return the index of the signal in the transmission (1-indexed) where `searchSignal` is a prefix of this signal. If `searchSignal` is a prefix of more than one signal, return the index of the first signal (minimum index). If there is no such signal, return `-1`.

A prefix of a string `s` is any leading contiguous substring of `s`.

```
def is_prefix_of_signal(transmission, searchSignal):
  pass
```

Example Usage:

```
print(is_prefix_of_signal("i love eating burger", "burg"))
print(is_prefix_of_signal("this problem is an easy problem", "pro"))
print(is_prefix_of_signal("i am tired", "you"))
```

Example Output:

```
4
2
-1
```

Close Section

▼ Advanced Problem Set Version 1

Problem 1: Blueprint Approval Process

You are in charge of overseeing the blueprint approval process for various architectural designs. Each blueprint has a specific complexity level, represented by an integer. Due to the complex nature of the designs, the approval process follows a strict order:

1. Blueprints with lower complexity should be reviewed first.
2. If a blueprint with higher complexity is submitted, it must wait until all simpler blueprints have been approved.

Your task is to simulate the blueprint approval process using a queue. You will receive a list of blueprints, each represented by their complexity level in the order they are submitted. Process the blueprints such that the simpler designs (lower numbers) are approved before more complex ones.

Return the order in which the blueprints are approved.

```
def blueprint_approval(blueprints):
    pass
```

Example Usage:

```
print(blueprint_approval([3, 5, 2, 1, 4]))
print(blueprint_approval([7, 4, 6, 2, 5]))
```

Example Output:

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

Problem 2: Build the Tallest Skyscraper

You are given an array `floors` representing the heights of different building floors. Your task is to design a skyscraper using these floors, where each floor must be placed on top of a floor with equal or greater height. However, you can only start a new skyscraper when necessary, meaning when no more floors can be added to the current skyscraper according to the rules.

Return the number of skyscrapers you can build using the given floors.

```
def build_skyscrapers(floors):
    pass
```

Example Usage:

```
print(build_skyscrapers([10, 5, 8, 3, 7, 2, 9]))
print(build_skyscrapers([7, 3, 7, 3, 5, 1, 6]))
print(build_skyscrapers([8, 6, 4, 7, 5, 3, 2]))
```

Example Output:

```
4
4
2
```

Problem 3: Dream Corridor Design

You are an architect designing a corridor for a futuristic dream space. The corridor is represented by a list of integer values where each value represents the width of a segment of the corridor. Your goal is to find two segments such that the corridor formed between them (including the two segments) has the maximum possible area. The area is defined as the minimum width of the two segments multiplied by the distance between them.

You need to return the maximum possible area that can be achieved.

```
def max_corridor_area(segments):
    pass
```

Example Usage:

```
print(max_corridor_area([1, 8, 6, 2, 5, 4, 8, 3, 7]))
print(max_corridor_area([1, 1]))
```

Example Output:

```
49
1
```

Problem 4: Dream Building Layout

You are an architect tasked with designing a dream building layout. The building layout is represented by a string `s` of even length `n`. The string consists of exactly `n / 2` left walls `'['` and `n / 2` right walls `']'`.

A layout is considered balanced if and only if:

- It is an empty space, or
- It can be divided into two separate balanced layouts, or
- It can be surrounded by left and right walls that balance each other out.

You may swap the positions of any two walls any number of times.

Return the minimum number of swaps needed to make the building layout balanced.

```
def min_swaps(s):
    pass
```

Example Usage:

```
print(min_swaps("][]["))
print(min_swaps("]]][[["))
print(min_swaps("[]"))
```

Example Output:

```
1
2
0
```

Problem 5: Designing a Balanced Room

You are designing a room layout represented by a string `s` consisting of walls `'('`, `')'`, and decorations in the form of lowercase English letters.

Your task is to remove the minimum number of walls `'('` or `')'` in any positions so that the resulting room layout is balanced and return any valid layout.

Formally, a room layout is considered balanced if and only if:

- It is an empty room (an empty string), contains only decorations (lowercase letters), or
- It can be represented as AB (A concatenated with B), where A and B are valid layouts, or
- It can be represented as (A), where A is a valid layout.

```
def make_balanced_room(s):
    pass
```

Example Usage:

```
print(make_balanced_room("art(t(d)e)sign)"))
print(make_balanced_room("d)e(s)ign"))
print(make_balanced_room("))(("))
```

Example Output:

```
art(t(d)e)s)ign
de(s)ign
```

Problem 6: Time to Complete Each Dream Design

As an architect, you are working on a series of imaginative designs for various dreamscapes. Each design takes a certain amount of time to complete, depending on the complexity of the elements involved. You want to know how many days it will take for each design to be ready for the next one to begin, assuming each subsequent design is more complex and thus takes more time to finish.

You are given an array `design_times` where each element represents the time in days needed to complete a particular design. For each design, determine the number of days you will have to wait until a more complex design (one that takes more days) is ready to begin. If no such design exists for a particular design, return `0` for that position.

Return an array `answer` such that `answer[i]` is the number of days you have to wait after the `i`-th design to start working on a more complex design. If there is no future design that is more complex, keep `answer[i] == 0` instead.

```
def time_to_complete_dream_designs(design_times):
    pass
```

Example Usage:

```
print(time_to_complete_dream_designs([3, 4, 5, 2, 1, 6, 7, 3]))
print(time_to_complete_dream_designs([2, 3, 1, 4]))
print(time_to_complete_dream_designs([5, 5, 5, 5]))
```

Example Output:

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

Problem 7: Next Greater Element

You are designing a sequence of dream elements, each represented by a number. The sequence is circular, meaning that the last element is followed by the first. Your task is to determine the next greater dream element for each element in the sequence.

The next greater dream element for a dream element `x` is the first element that is greater than x when traversing the sequence in its natural circular order. If no such dream element exists, return -1 for that dream element.

```
def next_greater_dream(dreams):
    pass
```

Example Usage:

```
print(next_greater_dream([1, 2, 1]))
print(next_greater_dream([1, 2, 3, 4, 3]))
```

Example Output:

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

Close Section

▼ Advanced Problem Set Version 2

Problem 1: Score of Mystical Market Chains

In the mystical market, chains of magical items are represented by a string of balanced symbols. The score of these chains is determined by the mystical power within the string, following these rules:

- The symbol `"()"` represents a basic magical item with a power score of 1.
- A chain `AB`, where `A` and `B` are balanced chains of magical items, has a total power score of `A + B`.
- A chain (`A`), where `A` is a balanced chain of magical items, has a power score of `2 * A`.

Given a balanced string representing a chain of magical items, return the total power score of the chain.

```
def score_of_mystical_market_chains(chain):
  pass
```

Example Usage:

```
print(score_of_mystical_market_chains("()"))
print(score_of_mystical_market_chains("(())"))
print(score_of_mystical_market_chains("()()"))
```

Example Output:

```
1
2
2
```

Problem 2: Arrange Magical Orbs

In the mystical market, you have a collection of magical orbs, each of which is colored red, white, or blue. Your task is to arrange these orbs in a specific order so that all orbs of the same color are adjacent to each other. The colors should be ordered as red, white, and blue.

We will use the integers 0, 1, and 2 to represent the colors red, white, and blue, respectively.

You must arrange the orbs in-place without using any library's sorting function.

```
def arrange_magical_orbs(orbs):
  pass
```

Example Usage:

```
orbs1 = [2, 0, 2, 1, 1, 0]
arrange_magical_orbs(orbs1)
print(orbs1)

orbs2 = [2, 0, 1]
arrange_magical_orbs(orbs2)
print(orbs2)
```

Example Output:

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

Problem 3: Matching of Buyers with Sellers

In the mystical market, you are given a list of buyers, where each buyer has a specific amount of gold to spend. You are also given a list of sellers, where each seller has a specific price for their magical goods.

A buyer can purchase from a seller if the buyer's gold is greater than or equal to the seller's price. Additionally, each buyer can make at most one purchase, and each seller can sell their goods to at most one buyer.

Return the maximum number of transactions that can be made in the market that satisfy these conditions.

```
def match_buyers_and_sellers(buyers, sellers):
  pass
```

Example Usage:

```
buyers1 = [4, 7, 9]
sellers1 = [8, 2, 5, 8]
print(match_buyers_and_sellers(buyers1, sellers1))

buyers2 = [1, 1, 1]
sellers2 = [10]
print(match_buyers_and_sellers(buyers2, sellers2))
```

Example Output:

```
3
0
```

Problem 4: Maximum Value from Removing Rare Items

In the Mystical Market, you are given a collection of mystical items in a string format `items` and two integers `x` and `y`. You can perform two types of operations any number of times to remove rare item pairs and gain value.

- Remove the pair of items `"AB"` and gain `x` value points.
- Remove the pair of items `"BA"` and gain `y` value points.

Return the maximum value you can gain after applying the above operations on items.

```
def maximum_value(items, x, y):
  pass
```

Example Usage:

```
s1 = "cdbcbbaaabab"
x1, y1 = 4, 5
print(maximum_value(s1, x1, y1))

s2 = "aabbaaxybbaabb"
x2, y2 = 5, 4
print(maximum_value(s2, x2, y2))
```

Example Output:

```
19
20
```

Problem 5: Strongest Magical Artifacts

In the Mystical Market, you are given an array of magical artifacts represented by integers `artifacts`, and an integer `k`.

A magical artifact `artifacts[i]` is said to be stronger than `artifacts[j]` if `|artifacts[i] - m| > |artifacts[j] - m|` where `m` is the median strength of the artifacts. If `|artifacts[i] - m| == |artifacts[j] - m|`, then `artifacts[i]` is said to be stronger than `artifacts[j]` if `artifacts[i] > artifacts[j]`.

Return a list of the strongest `k` magical artifacts in the Mystical Market. Return the answer in any arbitrary order.

```
def get_strongest_artifacts(artifacts, k):
  pass
```

Example Usage:

```
print(get_strongest_artifacts([1, 2, 3, 4, 5], 2))
print(get_strongest_artifacts([1, 1, 3, 5, 5], 2))
print(get_strongest_artifacts([6, 7, 11, 7, 6, 8], 5))
```

Example Output:

```
[5, 1]
[5, 5]
[11, 8, 6, 6, 7]
```

Problem 6: Enchanted Boats

You are given an array creatures where `creatures[i]` is the magical power of the `i`th creature, and an infinite number of enchanted boats where each boat can carry a maximum magical load of `limit`. Each boat carries at most two creatures at the same time, provided the sum of the magical power of those creatures is at most `limit`.

Return the minimum number of enchanted boats required to carry every magical creature.

```
def num_enchanted_boats(creatures, limit):
  pass
```

Example Usage:

```
print(num_enchanted_boats([1, 2], 3))
print(num_enchanted_boats([3, 2, 2, 1], 3))
print(num_enchanted_boats([3, 5, 3, 4], 5))
```

Example Output:

```
1
3
4
```

Problem 7: Market Token Value

You are a vendor in a mystical market where magical tokens are used for trading. The value of a token is determined by its structure, represented by a string containing pairs of mystical brackets `()`.

The value of a mystical token is calculated based on the following rules:

- `()` has a value of 1.
- The value of two adjacent tokens `AB` is the sum of their individual values, where `A` and `B` are valid token structures.
- The value of a nested token `(A)` is twice the value of the token inside the brackets.

Your task is to calculate the total value of a given mystical token string.

```
def token_value(token):
  pass
```

Example Usage:

```
print(token_value("()"))
print(token_value("(())"))
print(token_value("()()"))
```

Example Output:

```
1
2
2
```

Close Section