Data structure and Algorithms :

- **Why Do We Have So Many Algorithms and Data Structures?**
- **It's Essential to Understand When and How to Use Each Data Structure Effectively.**
- **What is a good data structure, and how do you determine its suitability for a specific problem?**
- **Why is balancing data structures important, and how does it impact efficiency?**
- **Can you provide an example of when using an unbalanced data structure led to inefficiency?**
- **How does the choice of data structure influence the overall design of an algorithm?**

Here is a concise overview of real-life examples of data structures and their applications:

1. Array

- **Use:** Storing elements of the same type in contiguous memory locations.
- **Examples:**
  - **Storing temperatures of a city over a week.**
  - Representing pixels in an image.

---

2. Linked List

- **Use:** Dynamically managing collections where frequent insertions and deletions occur.
- **Examples:**
  - **Music playlist (next and previous song navigation).**
  - Undo functionality in text editors.

---

3. Stack

- **Use:** LIFO (Last-In-First-Out) operations.
- **Examples:**
  - **Browser history (backward navigation).**
  - Expression evaluation (e.g., postfix or infix conversion).
  - Undo/Redo operations.

4. Queue

- **Use:** FIFO (First-In-First-Out) operations.
- **Examples:**
  - Print job scheduling in printers.
  - **Order processing in restaurants.**

---

7. Hash Table

- **Use:** Fast lookups using key-value pairs.
- **Examples:**
  - Dictionary implementation in programming languages.
  - **Storing user credentials in a database for quick authentication.**

---

8. Tree

- **Use:** Hierarchical data representation.
- **Examples:**
  - **Folder/directory structures in file systems.**
  - Organization charts in companies.
  - Parsing expressions in compilers.

---

9. Binary Search Tree (BST)

- **Use:** Efficient searching, insertion, and deletion.
- **Examples:**
  - Auto-suggestions in search engines.
  - Dynamic database indexing.

---

10. Graph

- **Use:** Representing connections or relationships.
- **Examples:**
  - Social networks (LinkedIn connections).
  - **GPS and Google Maps for shortest path navigation**

11. Set

- **Use:** Unique collection of elements, no duplicates.
- **Examples:**
  - Storing unique usernames.
  - Removing duplicates from a dataset.

Here are the types of problems we face that require using various data structures in the modern era, explained point-wise:

1. Efficient Data Management

- **Problem:** Large-scale data storage and retrieval, especially for real-time applications.
- **Solution:** Use **hash tables** or **dictionaries** for fast lookups or **databases** for structured data.
- **Example:** Managing user credentials in web applications.

2. Real-Time Processing

- **Problem:** Handling tasks that need immediate attention or execution in the correct order.
- **Solution:** Use **queues** or **priority queues** for task scheduling.
- **Example:** Processing print jobs, task scheduling in operating systems.

3. Network Representation

- **Problem:** Modeling and analyzing relationships or connections between entities.
- **Solution:** Use **graphs** to represent networks.
- **Example:** Social media friend suggestions, GPS navigation, or web page ranking in search engines.

---

4. Memory Constraints :

- **Problem:** Storing data in memory-efficient ways when resources are limited.
- **Solution:** Use **linked lists**, **binary trees**, to reduce memory overhead.
- **Example:** Implementing autocomplete in search engines or memory-efficient representation of strings.

---

5. Searching and Sorting

- **Problem:** Finding elements in large datasets or maintaining an ordered collection.
- **Solution:** Use **binary search trees**, **heaps**, or **arrays**.
- **Example:** Database indexing or finding the top-k trending topics.

---

6. Backtracking and Undo Operations

- **Problem:** Implementing functionalities that require going back to a previous state.
- **Solution:** Use **stacks** for Last-In-First-Out (LIFO) operations.
- **Example:** Undo/redo operations in editors or solving mazes using backtracking.

---

7. Data Deduplication

- **Problem:** Removing duplicates or ensuring data uniqueness.
- **Solution:** Use **sets** or **hash tables** for unique data storage.

- **Example:** Removing duplicate entries from a database or handling unique product IDs.

---

8. Dynamic Data Handling

- **Problem:** Handling data that grows or shrinks dynamically.
- **Solution:** Use **linked lists** or **dynamic arrays** (like Python's lists).
- **Example:** Managing playlists, browser history.

---

9. Handling Nested or Hierarchical Data

- **Problem:** Representing data with a parent-child relationship.
- **Solution:** Use **trees** or **graphs**.
- **Example:** Folder structures, organizational charts, or XML/JSON parsing.

---

10. Concurrent Access

- **Problem:** Allowing multiple users to access shared resources without conflict.
- **Solution:** Use **thread-safe queues** or **locks**.
- **Example:** Implementing producer-consumer problems in a multi-threaded system.

---

11. Optimizing Search Time

- **Problem:** Reducing the time taken to find specific data points in large datasets.
- **Solution:** Use **hash maps**, **tries**, or **binary search trees**.
- **Example:** Dictionary lookups, autocomplete, or keyword searching.

---

12. Pathfinding and Navigation

- **Problem:** Finding the shortest or most efficient path between two points.
- **Solution:** Use **graphs** and algorithms like Dijkstra or A*.

- **Example:** GPS navigation, network routing.

---

13. High-Performance Computing

- **Problem:** Performing computations efficiently on large datasets.
- **Solution:** Use **arrays**, **matrices**, or **heaps** for optimized operations.
- **Example:** Machine learning models or scientific simulations.

---

14. Managing Time-Based Data

- **Problem:** Handling time-dependent events or data streams.
- **Solution:** Use **queues** or **priority queues**.
- **Example:** Stock market monitoring, real-time messaging.

---

15. Predictive Analysis and Decision Making

- **Problem:** Storing and analyzing historical data for predictions.
- **Solution:** Use **graphs** and **trees** for structured data analysis.
- **Example:** Predicting customer behavior or modeling decision trees.

Here are the key issues if No data structure :

1. Inefficient Data Retrieval

- **Problem:** Searching for data in unstructured formats takes a lot of time.
- **Impact:** Operations like finding an element may require scanning the entire dataset (e.g., O(n) complexity instead of O(1) or O(log n)).
- **Example:** Searching for a specific file in an unsorted folder with no indexing.

---

2. High Memory Usage

- **Problem:** Without proper data structures, memory usage cannot be optimized.
- **Impact:** Redundant or unused data may consume large amounts of memory unnecessarily.
- **Example:** Storing duplicate elements without using a set or hash table.

---

3. Poor Code Organization

- **Problem:** Managing unstructured data makes the code messy and hard to maintain.
- **Impact:** Debugging, extending, or updating the code becomes challenging.
- **Example:** Storing data in flat files instead of using hierarchical structures like trees.

---

4. Lack of Scalability

- **Problem:** Handling large-scale data becomes impractical without efficient storage and processing.
- **Impact:** The application slows down or crashes under heavy loads.
- **Example:** A social media platform not using graphs for managing user connections.

---

5. Increased Redundancy

- **Problem:** Without structures like sets or hash maps, handling unique data becomes inefficient.
- **Impact:** Duplicate data increases redundancy, leading to wastage of resources.
- **Example:** Allowing duplicate IDs in a database.

---

6. Difficulty in Managing Complex Relationships

- **Problem:** Representing relationships between data points becomes difficult.

- **Impact:** Applications fail to represent hierarchical or interconnected data efficiently.
- **Example:** Not using graphs to model networks or trees for hierarchical data like folder structures.

---

7. Poor Performance in Real-Time Systems

- **Problem:** Real-time operations like scheduling or task handling become slow.
- **Impact:** Delays in execution affect system reliability and user experience.
- **Example:** Not using priority queues for task scheduling in operating systems.

---

8. Inability to Handle Dynamic Data

- **Problem:** Managing data that grows or shrinks dynamically becomes cumbersome.
- **Impact:** Memory leaks or crashes due to inefficient handling.
- **Example:** Not using linked lists for dynamic collections.

---

9. Limited Functionality

- **Problem:** Without data structures, some operations are impractical or impossible to implement efficiently.
- **Impact:** The application lacks essential features like autocomplete or search suggestions.
- **Example:** Not using tries for efficient prefix matching in search engines.

---

10. Increased Algorithm Complexity

- **Problem:** Algorithms become complex and slow when data is unstructured.
- **Impact:** The overall system performance decreases due to high time complexity.
- **Example:** Sorting a dataset without using efficient structures like heaps or trees.

---

11. Difficult Backtracking or Undo Operations

- **Problem:** Undo or rollback mechanisms become inefficient without proper structures.
- **Impact:** Systems may fail to restore previous states quickly.
- **Example:** Not using stacks for undo operations in text editors.

---

12. Inefficient Pathfinding and Navigation

- **Problem:** Finding the shortest path in networks becomes challenging.
- **Impact:** Navigation systems or routing algorithms are unable to provide optimal solutions.
- **Example:** Not using graphs for GPS navigation or network routing.

---

13. Unmanageable Concurrency

- **Problem:** Handling multiple users accessing shared data simultaneously becomes chaotic.
- **Impact:** Race conditions or deadlocks occur frequently.
- **Example:** Not using thread-safe queues for producer-consumer problems.

---

14. Redundant Calculations

- **Problem:** Without memoization or efficient caching, repetitive calculations are performed.
- **Impact:** System efficiency is drastically reduced.
- **Example:** Not using hash maps for memoization in dynamic programming.

---

15. Poor User Experience

- **Problem:** Slow response times and lack of features degrade the user experience.
- **Impact:** Users may abandon the application or system.

- **Example:** Slow search functionality due to the absence of index-based data retrieval.

Real-life examples of where specific data structures are used, along with the types of problems they solve easily:

Data Structure	Real-Life Example	Type of Problem It Solves
Array/List	- Product catalog on an e-commerce site (storing product IDs or names).	- Easy sequential storage and access to elements.
	- Leaderboard in a game (scores stored in order).	- Maintaining order or performing linear operations.
Set	- Maintaining a list of unique users logged into a website.	- Eliminates duplicates and checks membership efficiently.
	- Tracking unique IP addresses visiting a website.	- Handling unique elements quickly and easily.
Dictionary (HashMap)	- Phone book (mapping names to phone numbers).	- Provides fast key-based lookups, updates, and insertions.
	- Caching results of API calls for faster subsequent requests.	- Efficient association and retrieval of data.
Stack	- Browser history (backtracking to previous pages).	- Handles Last-In-First-Out (LIFO) scenarios like undo, backtracking, or nested operations.
	- Undo functionality in a text editor or image editing software.	- Reverting to the most recent action.
Queue	- Ticket booking system (handling customers in order of arrival).	- First-In-First-Out (FIFO) order processing, such as task scheduling or buffering.
	- Printing queue for multiple users sending jobs to the same printer.	- Processes tasks in order of arrival.

Deque	- Navigation systems (storing the most recent locations visited).	- Allows efficient addition/removal from both ends for flexible operations.
	- Task scheduling where tasks need to be dynamically added or removed from the beginning or end.	- Handles operations on both ends of the data efficiently.
Priority Queue (Heap)	- Event scheduling where high-priority events are handled first (e.g., emergency systems).	- Dynamically handles elements based on their priority, not insertion order.
	- Airline reservation systems (upgrading high-priority passengers).	- Provides access to the smallest/largest element efficiently.
Linked List	- Media players (forward and backward navigation in a playlist).	- Allows efficient insertion/deletion of elements without reallocating memory.
	- Managing memory blocks in operating systems (dynamic allocation).	- Efficient dynamic memory management and resizing.
Graph	- Google Maps for finding the shortest route between two locations.	- Models relationships and connectivity problems (e.g., paths, networks).
	- Social network friend suggestions (finding mutual friends).	- Solves problems involving connected entities and networks.
Tree	- File system hierarchy (folders and subfolders).	- Organizes hierarchical data for easy searching and traversal.
	- Organization charts in companies (employee hierarchy).	- Efficiently handles data requiring parent-child relationships.
Trie (Prefix Tree)	- Autocomplete feature in search engines or text editors.	- Quickly matches prefixes and performs efficient word lookups.
	- Spell checking in word processors.	- Handles prefix-based searching and matching efficiently.
Hash Table	- Storing session tokens in web applications for fast authentication.	- Provides constant-time average-case complexity for lookups, insertions, and deletions.

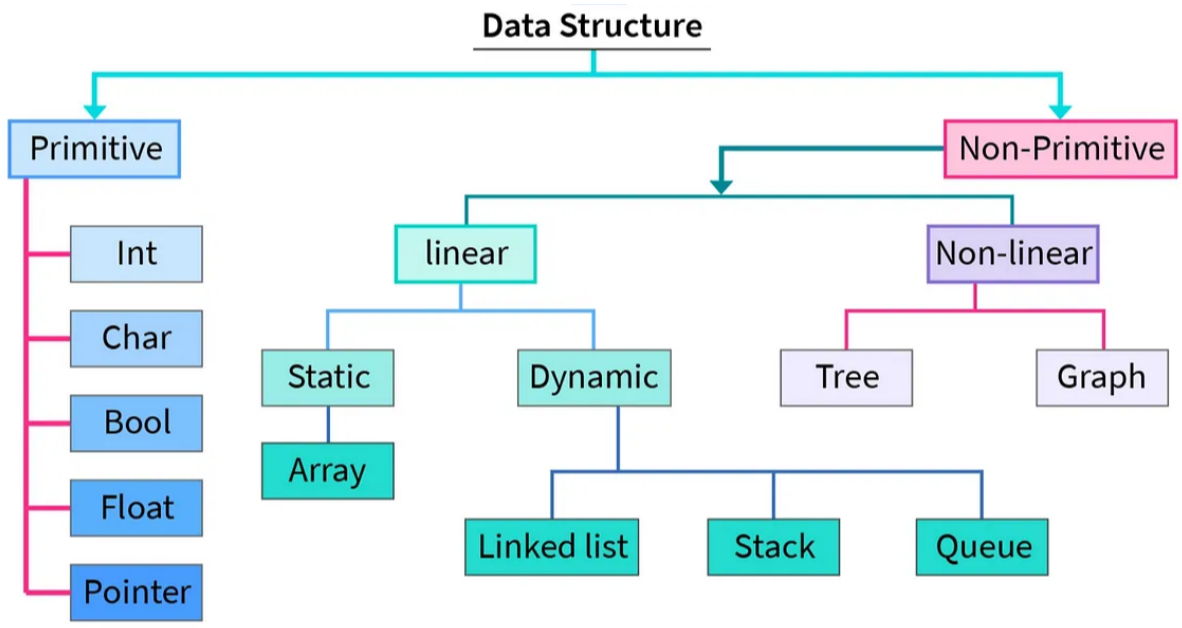
	- Storing and retrieving user preferences in applications.	- Efficiently handles key-value associations for quick access.

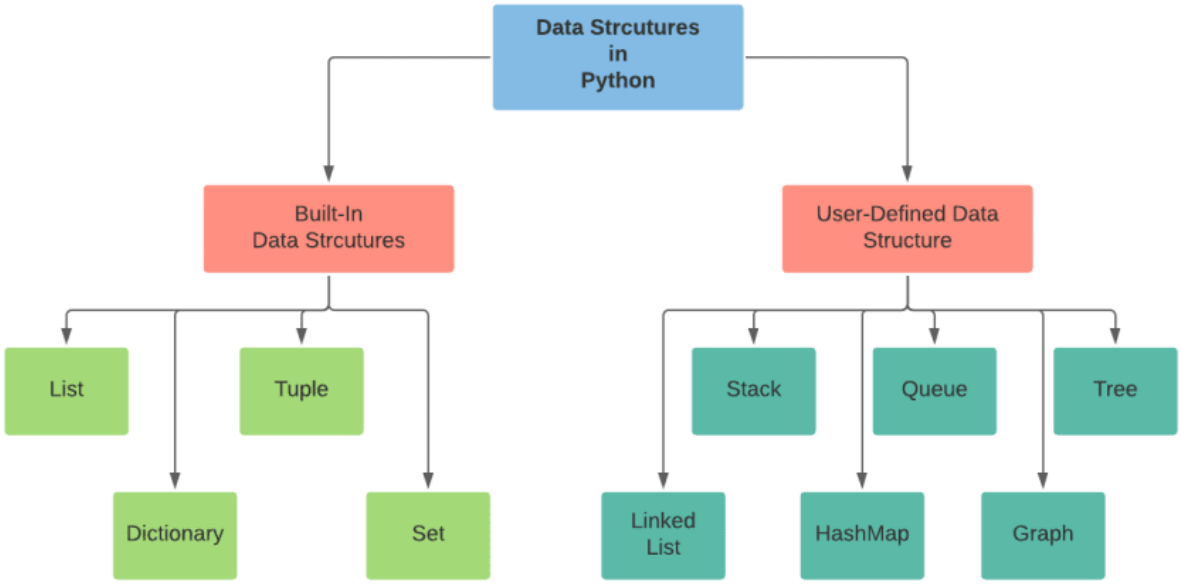
Algorithms and Their Real-Life Applications

Algorithm Name	Real-Life Problems Solved
Linear Search	- Finding a contact in an unsorted list.
	- Searching for a specific product in an inventory database.
Binary Search	- Searching for a word in a dictionary (sorted data).
	- Locating a file in a sorted directory structure.
Breadth-First Search (BFS)	- Finding the shortest path in an unweighted graph (e.g., navigation apps).
	- Analyzing social network connections.
Depth-First Search (DFS)	- Solving mazes or puzzles (e.g., Sudoku).
	- Navigating file systems to search for specific files.
Dijkstra’s Algorithm	- Finding the shortest path in weighted graphs (e.g., Google Maps for road navigation).
	- Routing data packets in a network efficiently.
A* Search Algorithm	- Pathfinding in video games (AI navigation).
	- Optimizing delivery routes for logistics companies.
Merge Sort	- Sorting customer names in alphabetical order.
	- Organizing large datasets in ascending or descending order.
Quick Sort	- Sorting a list of e-commerce products by price or rating.
	- Efficiently sorting database records.
Kruskal’s Algorithm	- Designing minimum-cost networks like roads, pipelines, or internet connections.
	- Building efficient electrical circuit layouts.
Prim’s Algorithm	- Planning network cabling for a city with minimal cost.

	- Designing railway tracks between cities.
Hashing	- Efficiently storing and retrieving passwords in databases.
	- Implementing cache mechanisms for faster data access.
Dynamic Programming (e.g., Knapsack)	- Resource allocation problems (e.g., maximizing profit with limited resources).
	- Planning investment portfolios.
Greedy Algorithms	- Scheduling task

Array:





1. Array Operations (using Python's `array` module)

a. Creation

- **Operation**: Create an array
- **Example**:

```
python
CopyEdit
import array
arr = array.array('i', [1, 2, 3, 4, 5])
```

b. Append

- **Operation**: Add an element at the end of the array.
- **Example**:

```
arr.append(6)
```

- **Time Complexity**: **O(1)** (amortized)

c. Insert

- **Operation**: Insert an element at a specific index.
- **Example**:

```
arr.insert(2, 10)
```

- **Time Complexity**: **O(n)** (since elements need to be shifted)

d. Pop

- **Operation**: Remove and return the last element.
- **Example**:

```
python
CopyEdit
arr.pop()
```

- **Time Complexity**: **O(1)**

e. Remove

- **Operation**: Remove the first occurrence of a value.
- **Example**:

```
python
CopyEdit
arr.remove(3)
```

- **Time Complexity**: **O(n)** (since the array needs to be traversed to find the element)

f. Access

- **Operation**: Access an element by index.
- **Example**:

```
python
CopyEdit
arr[2]
```

- **Time Complexity**: **O(1)**

g. Slice

- **Operation**: Get a subarray (slice) of the array.
- **Example**:

```
arr[1:3]
```

- **Time Complexity**: **O(k)** (where `k` is the size of the slice)

h. Reverse

- **Operation**: Reverse the array.
- **Example**:

```
arr.reverse()
```

- **Time Complexity**: **O(n)**

i. Extend

- **Operation**: Extend the array with another array.
- **Example**:

```
arr.extend([7, 8, 9])
```

- **Time Complexity**: **O(k)** (where `k` is the length of the array being added)
-

---

2. NumPy Array Operations

For performance reasons, **NumPy arrays** are commonly used for numerical computations and large-scale data handling. Here's a list of common NumPy array operations and their time complexities.

a. Array Creation

- **Operation**: Create a NumPy array
- **Example**:

```
python
CopyEdit
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

- **Time Complexity**: **O(n)**

b. Element-wise Operations

- **Operation**: Apply operations (e.g., addition, multiplication) element-wise on the array.
- **Example**:

```
python
CopyEdit
arr = np.array([1, 2, 3])
arr = arr * 2  # Element-wise multiplication
```

- **Time Complexity**: **O(n)**

c. Indexing/Accessing Elements

- **Operation**: Access an element by index.
- **Example**:

```
python
CopyEdit
arr[2]
```

- **Time Complexity**: **O(1)**

d. Slicing

- **Operation**: Slice a portion of the array.
- **Example**:

```
python
CopyEdit
```

```
arr[1:4]
```

- **Time Complexity**: **O(k)** (where `k` is the size of the slice)

e. Appending

- **Operation**: Add elements to the end of a NumPy array.
- **Example**:

```
python
CopyEdit
arr = np.append(arr, [6, 7])
```

- **Time Complexity**: **O(n)** (since NumPy arrays are fixed size and need to be resized)

f. Inserting

- **Operation**: Insert an element at a specific index.
- **Example**:

```
python
CopyEdit
arr = np.insert(arr, 2, 10)
```

- **Time Complexity**: **O(n)** (due to shifting elements)

g. Deleting

- **Operation**: Remove an element by index.
- **Example**:

```
python
CopyEdit
arr = np.delete(arr, 2)
```

- **Time Complexity**: **O(n)** (since elements must be shifted)

h. Reshaping

- **Operation**: Change the shape of the array.
- **Example**:

```
python
CopyEdit
arr = arr.reshape(3, 2)
```

- **Time Complexity**: **O(1)** (as long as the total number of elements remains the same)

i. Concatenation

- **Operation**: Concatenate two arrays.
- **Example**:

```
python
CopyEdit
arr2 = np.array([6, 7, 8])
arr = np.concatenate((arr, arr2))
```

- **Time Complexity**: **O(n + m)** (where `n` is the length of the first array, and `m` is the length of the second)

j. Sorting

- **Operation**: Sort an array.
- **Example**:

```
python
CopyEdit
arr.sort()
```

- **Time Complexity**: **O(n log n)** (for most sorting algorithms like QuickSort or MergeSort)

k. Broadcasting

- **Operation**: Apply operations to arrays with different shapes.
- **Example**:

```
python
CopyEdit
arr = np.array([1, 2, 3])
arr2 = np.array([10])
result = arr + arr2  # Broadcasting
```

- **Time Complexity**: **O(n)** (element-wise operation after broadcasting)

Summary of Time Complexities:

Operation	Array (Python)	NumPy Array
Creation	O(n)	O(n)
Append	O(1)	O(n)
Insert	O(n)	O(n)
Pop	O(1)	O(1)
Remove	O(n)	O(n)
Access	O(1)	O(1)

Slice	O(k)	O(k)
Reverse	O(n)	O(n)
Extend	O(k)	O(k)
Element-wise Ops	N/A	O(n)
Sorting	N/A	O(n log n)
Broadcasting	N/A	O(n)
Concatenation	N/A	O(n + m)

For Python arrays (from the `array` module), here's the table:

Operation	One-Dimensional Array	Two-Dimensional Array
Accessing an Element by Index	Time: O(1) Space: O(1)	Time: O(1) Space: O(1)
Inserting an Element at the End	Time: O(1) (Amortized) Space: O(1)	Time: O(1) (Amortized) Space: O(1)
Inserting an Element at the Beginning	Time: O(n) Space: O(n)	Time: O(m * n) Space: O(m * n)
Searching for an Element (Linear Search)	Time: O(n) Space: O(1)	Time: O(m * n) Space: O(1)
Transposing a Matrix	N/A	Time: O(m * n) Space: O(m * n)
Deletion of an Element	Time: O(n) Space: O(1)	Time: O(m * n) Space: O(1)

Explanation:

1. **Accessing an Element by Index**:
   - **One-Dimensional Array**: Accessing any element in a Python array is done in constant time (`O(1)`), as arrays are implemented as contiguous blocks in memory. Similarly, the space used is constant (`O(1)`).
   - **Two-Dimensional Array**: While Python doesn't have built-in two-dimensional arrays (i.e., an array of arrays), you can simulate this using nested lists or arrays, so the time complexity is still `O(1)` for direct access.
2. **Inserting an Element at the End**:

- **One-Dimensional Array**: Inserting an element at the end of the array is typically done in `O(1)` time (amortized). This is because, on average, appending an element takes constant time. The space complexity is `O(1)` as well.
- **Two-Dimensional Array**: Appending an element at the end of the outer array (or a new row) would also take `O(1)` time, provided the array has room to grow.

3. **Inserting an Element at the Beginning**:
   - **One-Dimensional Array**: Inserting an element at the beginning of the array requires shifting all elements, so it takes `O(n)` time. The space complexity is `O(n)` as well.
   - **Two-Dimensional Array**: For a 2D array (list of lists), inserting an element at the beginning would involve shifting all rows, leading to `O(m * n)` time, where `m` is the number of rows and `n` is the number of columns.
4. **Searching for an Element (Linear Search)**:
   - **One-Dimensional Array**: Searching for an element linearly requires checking every element, leading to `O(n)` time. Space complexity is `O(1)` as we don't use additional space other than the array itself.
   - **Two-Dimensional Array**: For a 2D array, searching requires checking each element across both dimensions (`m * n`), leading to `O(m * n)` time.
5. **Transposing a Matrix**:
   - **One-Dimensional Array**: A 1D array doesn't support transposition (i.e., converting a 1D structure into another 1D structure), so this operation doesn't apply.
   - **Two-Dimensional Array**: Transposing a matrix involves swapping rows with columns, which takes `O(m * n)` time and results in an `O(m * n)` space complexity, where `m` is the number of rows and `n` is the number of columns.
6. **Deletion of an Element**:
   - **One-Dimensional Array**: Deleting an element from an array involves shifting the remaining elements, so it takes `O(n)` time, with `O(1)` space

complexity.

- **Two-Dimensional Array**: Deletion from a 2D array involves shifting rows and possibly elements within the rows, so it has a time complexity of `O(m * n)`, but space complexity is `O(1)` as we only modify the array in-place.

Problems in Arrays

1. Fixed Size

- **Disadvantage**: Once an array is created, its size cannot be changed. To add or remove elements, a new array must be created.
- **Code Example**:

```
import array
arr = array.array('i', [1, 2, 3, 4])
# Cannot change size directly. Need to create a new array.
arr = arr + array.array('i', [5, 6])  # Concatenating arra
ys
```

2. Homogeneous Elements

- **Disadvantage**: Arrays can only store elements of the same data type. This limits their flexibility.
- **Code Example**:

```
# Error: You cannot mix data types in an array
arr = array.array('i', [1, 'two', 3])  # Will raise an err
or
```

3. Memory Inefficiency

- **Disadvantage**: Arrays may lead to memory inefficiency if the array is not fully utilized, especially if the size is large but only a few elements are used.
- **Example**: If you allocate an array of size 100, but only use 10 elements, the rest of the array consumes memory without being used.

4. Inefficient Insertions/Deletions

- **Disadvantage**: Inserting or deleting elements in the middle of the array is inefficient as it requires shifting elements, which can be slow for large arrays.
- **Code Example**:

```
arr = array.array('i', [1, 2, 3, 4])
arr.insert(2, 10)  # Insert 10 at index 2 (inefficient)
# Time complexity: O(n) because all elements after index 2
must be shifted
```

5. No Built-in Methods for Common Operations

- **Disadvantage**: The `array` module provides very limited built-in methods for operations like sorting, reversing, or other manipulations. You need to write custom code for many tasks.
- **Code Example**:

```
# No built-in sorting function for arrays
arr = array.array('i', [4, 2, 3, 1])
arr = sorted(arr)  # Works, but returns a list, not an arr
ay
```

6. Limited Functionality for Complex Operations

- **Disadvantage**: Arrays lack many advanced operations provided by more specialized data structures like lists or NumPy arrays (e.g., multidimensional arrays, element-wise operations).
- **Code Example** (array cannot handle broadcasting):

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([10])
result = arr1 + arr2  # Broadcasting works with NumPy but
not with array
```

7. Inefficient for Large Datasets

- **Disadvantage**: Arrays in Python (especially from the `array` module) are less efficient for large datasets compared to more advanced data structures like NumPy arrays or lists due to their limitations and lack of optimizations for numerical operations.
- **Example**: For large numerical datasets, NumPy provides much better performance and memory efficiency than arrays from the `array` module.

8. No Dynamic Resizing

- **Disadvantage**: In Python, arrays from the `array` module do not support dynamic resizing (like lists). If you need to frequently add or remove elements, it may be better to use a list or other data structures.
- **Code Example**:

```
arr = array.array('i', [1, 2, 3])
arr.append(4)  # Appending is possible, but resizing in-pl
ace is not.
```

9. Incompatibility with Non-Primitive Types

- **Disadvantage**: Arrays do not support storing objects or non-primitive types (e.g., lists of lists or dictionaries), making them less versatile than lists.
- **Code Example**:

```
arr = array.array('i', [[1, 2], [3, 4]])  # This will rais
e an error
```

10. No Built-in Support for Multidimensional Arrays

- **Disadvantage**: Arrays in the `array` module are one-dimensional. For multidimensional data, you need to implement your own handling or use libraries like NumPy.
- **Example**: If you want a 2D array, you need to manually manage the dimensions or use other libraries.

Why Do We Need Arrays When We Have Lists?

1. **Memory Efficiency:**
    - Arrays are more memory-efficient compared to lists. They store elements of the same data type, whereas lists can store elements of mixed types.
    - **Example:Output:**Arrays consume less memory because they enforce type consistency.

```
import sys

my_list = [1, 2, 3, 4, 5]  # List
```

```
print("Size of list:", sys.getsizeof(my_list))

from array import array
my_array = array('i', [1, 2, 3, 4, 5])  # Array
print("Size of array:", sys.getsizeof(my_array))
```

2. **Type Safety:**
   - Arrays enforce type consistency, meaning all elements must be of the same data type. This reduces the chances of errors when performing mathematical or logical operations.
   - **Example:**

```
from array import array

my_array = array('i', [1, 2, 3, 4])  # Integer array
my_array.append(5)  # Works fine
# my_array.append("hello")  # Raises TypeError because
it's not an integer
print(my_array)
```

3. **Performance:**
   - For numerical computations, arrays are faster because they avoid the overhead of type checking for each element during operations.
   - **Example:Output:**

```
import time
from array import array

# Using a list
my_list = list(range(1000000))
```

```
start = time.time()
sum_list = sum(my_list)
print("Time taken by list:", time.time() - start)

# Using an array
my_array = array('i', range(1000000))
start = time.time()
sum_array = sum(my_array)
print("Time taken by array:", time.time() - start)
```

4. **Specialized Use Cases:**
    - Arrays are better suited for numerical and scientific computations where all elements share the same type.
    - Libraries like **NumPy** use arrays as their core data structure for matrix operations, numerical integration, and more.
    - **Example with NumPy:**

```
import numpy as np

my_numpy_array = np.array([1, 2, 3, 4, 5])
print("Numpy Array:", my_numpy_array)
print("Mean:", np.mean(my_numpy_array))
print("Sum:", np.sum(my_numpy_array))
```

5. **List Flexibility vs. Array Specialization:**
    - Lists are more flexible and can store elements of different types.
    - Arrays are specialized for uniform data and are more efficient in computational and

Key Differences Between List and Array

Feature	List	Array
Type of Data	Can store mixed data types	Stores data of the same type only
Memory Usage	More memory overhead	Memory efficient
Performance	Slower for numerical computations	Faster for numerical computations
Operations	General-purpose operations	Specialized for numerical operations
Library Dependency	Built-in	Requires `array` or external libraries like NumPy

When to Use Arrays Instead of Lists?

- When you need **memory efficiency** for large datasets.
- When performing **mathematical computations**.
- When working with **type-consistent data** (e.g., integers, floats).
- When using **libraries like NumPy** for advanced scientific tasks.

1. Creating a 1D Array Using the `array` Module

To create an array in Python, you need to import the `array` module and use the `array()` constructor.

```
import array

# 'i' specifies signed integers
arr_int = array.array('i', [1, 2, 3])
print("Integer Array:", arr_int)

import numpy as np
```

```
# Create a 1D array
arr_1d = np.array([1, 2, 3, 4, 5])
print("1D Array:", arr_1d)
print("Data Type:", arr_1d.dtype

# Create a 1D array of inte
```

Here:

- `'i'` specifies that the array will hold integers.
- `[1, 2, 3, 4, 5]` is the list of elements that will form the array.

2. Creating a 2D Array Using `numpy`

While the `array` module in Python only supports 1D arrays, you can use `numpy` to create 2D arrays (and even higher-dimensional arrays).

```
import numpy as np

# Create a 2D array (matrix)
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("2D Array:\n", arr_2d)
```

Here:

- `np.array()` is used to create the array.
- `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]` represents a 2D array with 3 rows and 3 columns.

3. Basic Operations on Arrays

Let's go through some common array operations on both 1D and 2D arrays.

a. Accessing Elements

- **1D Array**: You can access elements using an index.

- **2D Array**: You access elements by row and column index.

```
# 1D Array Access
print("Element at index 2 (1D Array):", arr[2])  # Access ele
ment at index 2 (3rd element)

# 2D Array Access
print("Element at position (1, 2) (2D Array):", arr_2d[1][2])
# Access element at row 1, column 2
```

b. Traversing Arrays

- **1D Array**: Use a loop to traverse and print elements.
- **2D Array**: Use nested loops to traverse rows and columns.

```
# Traversing 1D Array
print("Traversing 1D Array:")
for element in arr:
    print(element, end=' ')

# Traversing 2D Array
print("\nTraversing 2D Array:")
for row in arr_2d:
    for element in row:
        print(element, end=' ')
    print()  # Newline after each row
```

c. Insertion

- **1D Array**: Use `insert()` to insert an element at a specific position.
- **2D Array**: You can use slicing to insert elements or append rows/columns.

```
# Insertion in 1D Array
arr.insert(2, 10)  # Insert 10 at index 2
print("1D Array after insertion:", arr)

# Insertion in 2D Array (Adding a new row)
new_row = [10, 11, 12]
arr_2d = np.vstack([arr_2d, new_row])  # Adds a new row to th
e 2D array
print("2D Array after inserting a new row:\n", arr_2d)
```

d. Deletion

- **1D Array**: Use `remove()` to remove an element or `pop()` to remove at a specific index.
- **2D Array**: Use `np.delete()` to remove rows or columns.

```
# Deletion in 1D Array
arr.remove(10)  # Removes the first occurrence of 10
print("1D Array after deletion:", arr)

# Deletion in 2D Array (Removing a row)
arr_2d = np.delete(arr_2d, 1, axis=0)  # Removes row at index
1
print("2D Array after deleting a row:\n", arr_2d)
```

e. Updating Elements

- **1D Array**: Directly assign a new value at a specific index.
- **2D Array**: Update using row and column indices.

```
# Updating in 1D Array
arr[1] = 20  # Update element at index 1 to 20
print("1D Array after updating:", arr)

# Updating in 2D Array
arr_2d[1][1] = 50  # Update element at row 1, column 1 to 50
print("2D Array after updating:\n", arr_2d)
```

f. Slicing Arrays

- **1D Array**: Use slice notation `arr[start:end]`.
- **2D Array**: Use slice notation for rows and columns.

```
# Slicing in 1D Array
sub_arr_1d = arr[1:4]  # Get elements from index 1 to 3
print("Sliced 1D Array:", sub_arr_1d)

# Slicing in 2D Array (getting a sub-matrix)
sub_arr_2d = arr_2d[0:2, 1:3]  # Get elements from rows 0 and
1, columns 1 to 2
print("Sliced 2D Array:\n", sub_arr_2d)
```

4. Some Additional Operations

a. Sorting

- **1D Array**: Use the `sorted()` function or `arr.sort()`.
- **2D Array**: Use `numpy.sort()`.

```
# Sorting 1D Array
```

```
arr_sorted = sorted(arr)  # Returns a sorted list (does not m
odify original)
print("Sorted 1D Array:", arr_sorted)

# Sorting 2D Array by a specific row (e.g., row 0)
arr_2d_sorted = np.sort(arr_2d, axis=1)  # Sort along columns
print("Sorted 2D Array:\n", arr_2d_sorted)
```

b. Reversing

- **1D Array**: Use slicing `[::-1]`.
- **2D Array**: Use `np.flip()`.

```
python
Copy code
# Reversing 1D Array
arr_reversed = arr[::-1]  # Reverse the array
print("Reversed 1D Array:", arr_reversed)

# Reversing 2D Array
arr_2d_reversed = np.flip(arr_2d, axis=0)  # Reverse along ro
ws
print("Reversed 2D Array:\n", arr_2d_reversed)
```

Operations:

---

1. Traversal: Looping Through Arrays Using `for` Loop

Looping through an array allows you to access each element in the array.

```
python
Copy code
import array

arr = array.array('i', [1, 2, 3, 4, 5])

# Traversing through the array
for elem in arr:
    print(elem)
```

2. Insertion: Insert Elements at Specific Positions Using `insert()` or Array Slicing

- **Using** `insert()`: Adds an element at a specified position.
- **Using array slicing**: Adds elements at the specified index by slicing.

```
python
Copy code
import array

arr = array.array('i', [1, 2, 3, 4])

# Insert element at position 2 (before the element 3)
arr.insert(2, 99)
print("Array after insertion:", arr)

# Using slicing to insert
arr = arr[:2] + array.array('i', [88]) + arr[2:]
print("Array after insertion using slicing:", arr)
```

3. Deletion: Deleting Elements Using `remove()` or `pop()`

- **Using** `remove()`: Removes the first occurrence of a specified value.
- **Using** `pop()`: Removes and returns the element at a specified position.

```
python
Copy code
import array

arr = array.array('i', [1, 2, 3, 4, 5])

# Remove the first occurrence of value 3
arr.remove(3)
print("Array after removal of 3:", arr)

# Pop the element at position 2 (indexing starts at 0)
popped_element = arr.pop(2)
print("Popped element:", popped_element)
print("Array after popping:", arr)
```

4. Updating Elements: Modifying Elements in the Array

You can update elements by directly accessing them using their index.

```
import array

arr = array.array('i', [1, 2, 3, 4, 5])

# Update element at index 1 (change value 2 to 10)
arr[1] = 10
print("Array after updating index 1:", arr)
```

5. Array Slicing: Extracting a Subarray Using Slicing

Array slicing allows you to create a subarray (a new array) by specifying a range of indices.

```
import array

arr = array.array('i', [1, 2, 3, 4, 5, 6])

# Extracting a subarray from index 2 to 4 (excluding 5)
subarray = arr[2:5]
print("Subarray from index 2 to 4:", subarray)

# Extracting elements from the beginning up to index 3
subarray2 = arr[:3]
print("Subarray from the start to index 3:", subarray2)

# Extracting elements from index 3 to the end
subarray3 = arr[3:]
print("Subarray from index 3 to the end:", subarray3)
```

Summary:

- **Traversal**: Loop through the array using a `for` loop to access each element.
- **Insertion**: Use `insert()` or slicing to add elements at specific positions.
- **Deletion**: Use `remove()` to delete specific elements or `pop()` to remove an element by index.
- **Updating Elements**: Modify elements by accessing them directly using their index.
- **Array Slicing**: Extract subarrays using slicing (specifying a range of indices).

1. Common Methods

a. `append()` : Adds an element at the end of the array.

```
python
Copy code
import array

arr = array.array('i', [1, 2, 3])
arr.append(4)  # Add 4 to the end of the array
print("Array after append:", arr)
```

b. `extend()` : Adds multiple elements at the end of the array.

```
python
Copy code
arr = array.array('i', [1, 2, 3])
arr.extend([4, 5, 6])  # Add multiple elements
print("Array after extend:", arr)
```

c. `pop()` : Removes and returns the element at a specified index. If no index is provided, it removes and returns the last element.

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4])
popped_element = arr.pop()  # Remove and return the last elem
ent (4)
print("Popped element:", popped_element)
print("Array after pop:", arr)
```

d. `remove()` : Removes the first occurrence of a specified value.

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4, 3])
arr.remove(3)  # Remove the first occurrence of 3
print("Array after remove:", arr)
```

e. `index()`: Returns the index of the first occurrence of a specified value.

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4])
index_of_3 = arr.index(3)  # Find the index of element 3
print("Index of 3:", index_of_3)
```

2. Sorting and Reversing

a. Sorting: Sort the array in ascending or descending order.

```
python
Copy code
arr = array.array('i', [5, 3, 8, 1, 7])

# Sort in ascending order
arr = sorted(arr)  # returns a sorted list
print("Sorted array in ascending order:", arr)

# Sort in descending order
arr = sorted(arr, reverse=True)
print("Sorted array in descending order:", arr)
```

Note: The `array` module doesn't directly support sorting in place like lists do (`arr.sort()`), so `sorted()` is used here.

b. Reversing: Reverses the order of elements in the array.

```
python
Copy code
arr = array.array('i', [1, 2, 3, 4, 5])
arr = arr[::-1]  # Reversing the array using slicing
print("Reversed array:", arr)
```

3. Array Conversion

a. Converting Array to List: Convert an array to a Python list using `list()`.

```
arr = array.array('i', [1, 2, 3, 4])
arr_list = list(arr)  # Convert array to list
print("Array converted to list:", arr_list)
```

b. Converting List to Array: Convert a list to an array using the `array()` constructor.

Questions :

1. Find the Largest/Smallest Element in an Array

Problem: Given an array, find the largest and smallest elements.

Approach:

- Iterate through the array and compare each element with the current largest/smallest element.

```
python
Copy code
def find_largest_smallest(arr):
    largest = max(arr)
    smallest = min(arr)
    return largest, smallest

arr = [12, 45, 67, 2, 89, 34]
largest, smallest = find_largest_smallest(arr)
print(f"Largest: {largest}, Smallest: {smallest}")
```

Output:

```
yaml
Copy code
Largest: 89, Smallest: 2
```

---

2. Find Duplicates in an Array

Problem: Given an array, find all the elements that appear more than once.

Approach:

- Use a set to keep track of elements that have been seen before. If you encounter an element that is already in the set, it's a duplicate.

```
def find_duplicates(arr):
    seen = set()
    duplicates = []
    for num in arr:
        if num in seen:
            duplicates.append(num)
        else:
            seen.add(num)
    return duplicates

arr = [1, 2, 3, 2, 4, 5, 1]
duplicates = find_duplicates(arr)
print("Duplicates:", duplicates)
```

Output:

```
makefile
Copy code
Duplicates: [2, 1]
```

---

3. Reverse an Array

Problem: Reverse the elements of the array.

Approach:

- Use Python slicing or a loop to reverse the array in place.

```
def reverse_array(arr):
    return arr[::-1]

arr = [1, 2, 3, 4, 5]
```

```
reversed_arr = reverse_array(arr)
print("Reversed Array:", reversed_arr)
```

Output:

```
Reversed Array: [5, 4, 3, 2, 1]
```

---

4. Remove Duplicate Elements

Problem: Remove duplicates from an array.

Approach:

- Use a set to track unique elements and eliminate duplicates.

```
python
Copy code
def remove_duplicates(arr):
    return list(set(arr))

arr = [1, 2, 3, 2, 4, 5, 1]
unique_arr = remove_duplicates(arr)
print("Array after removing duplicates:", unique_arr)
```

```
```

Note: The order of elements may not be preserved when converting to a set. To maintain the order:

```
python
Copy code
def remove_duplicates(arr):
```

```
    seen = set()
    result = []
    for num in arr:
        if num not in seen:
            result.append(num)
            seen.add(num)
    return result

unique_arr = remove_duplicates(arr)
print("Array after removing duplicates (maintaining order):",
unique_arr)
```

5. Rotate an Array

Problem: Rotate an array by `k` positions to the right.

Approach:

- U

```
def rotate_array(arr, k):
    n = len(arr)
    k = k % n  # Handle cases where k > n
    return arr[-k:] + arr[:-k]

arr = [1, 2, 3, 4, 5]
rotated_arr = rotate_array(arr, 2)
print("Array after rotating by 2 positions:", rotated_arr)
```

Output:

```
Array after rotating by 2 positions: [4, 5, 1, 2, 3]
```

6. Merge Two Sorted Arrays

Problem: Given two sorted arrays, merge them into a single sorted array.

Approach:

- Use two pointers to iterate through both arrays and add the smaller element to the result array.

```
def merge_arrays(arr1, arr2):
    result = []
    i, j = 0, 0
    while i < len(arr1) and j < len(arr2):
        if arr1[i] < arr2[j]:
            result.append(arr1[i])
            i += 1
        else:
            result.append(arr2[j])
            j += 1
    result.extend(arr1[i:])
    result.extend(arr2[j:])
    return result

arr1 = [1, 3, 5, 7]
arr2 = [2, 4, 6, 8]
merged_arr = merge_arrays(arr1, arr2)
print("Merged Array:", merged_arr)
```

Output:

```
javascript
Copy code
```

```
Merged Array: [1, 2, 3, 4, 5, 6, 7, 8]
```

---

7. Find Intersection of Two Arrays

Problem: Find the common elements (intersection) between two arrays.

Approach:

- Convert both arrays to sets and find their intersection.

```
python
Copy code
def intersection(arr1, arr2):
    return list(set(arr1) & set(arr2))

arr1 = [1, 2, 3, 4, 5]
arr2 = [4, 5, 6, 7]
common_elements = intersection(arr1, arr2)
print("Intersection:", common_elements)
```

Here are the best time complexity solutions for each of the given problems:

1. Move All Zeroes to End

The task is to move all the zeroes in an array to the end while maintaining the order of non-zero elements.

Best Approach: **O(n)** time complexity with **O(1)** space complexity using the two-pointer technique.

```
python
CopyEdit
def move_zeroes_to_end(arr):
    non_zero_index = 0  # Pointer for the next non-zero eleme
nt
```

```
    for i in range(len(arr)):
        if arr[i] != 0:
            # Swap non-zero element to the front
            arr[non_zero_index], arr[i] = arr[i], arr[non_zer
o_index]
            non_zero_index += 1

    return arr

# Example
arr = [0, 1, 9, 0, 3, 12, 0]
print(move_zeroes_to_end(arr))  # Output: [1, 9, 3, 12, 0, 0,
0]
```

Explanation:

- **Time Complexity**: O(n), where `n` is the length of the array. We are iterating through the array only once.
- **Space Complexity**: O(1), as we are using only constant extra space (a few pointers).

---

2. Reverse Array in Groups

The task is to reverse the array in groups of a given size `k`.

Best Approach: **O(n)** time complexity with **O(1)** space complexity.

```
python
CopyEdit
def reverse_in_groups(arr, k):
    for i in range(0, len(arr), k):
        # Reverse the subarray of size k
        arr[i:i+k] = arr[i:i+k][::-1]

    return arr
```

```
# Example
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
k = 3
print(reverse_in_groups(arr, k))  # Output: [3, 2, 1, 6, 5,
4, 9, 8, 7]
```

Explanation:

- **Time Complexity**: O(n), where `n` is the length of the array. We reverse subarrays of size `k` in linear time.
- **Space Complexity**: O(1), as we are reversing the array in-place without using extra space.

---

3. Rotate Array

The task is to rotate the array by `d` elements to the right.

Best Approach: **O(n)** time complexity with **O(1)** space complexity.

```
python
CopyEdit
def rotate_array(arr, d):
    n = len(arr)
    # Handle the rotation using slice notation
    arr[:] = arr[d % n:] + arr[:d % n]
    return arr

# Example
arr = [1, 2, 3, 4, 5, 6, 7]
d = 3
print(rotate_array(arr, d))  # Output: [5, 6, 7, 1, 2, 3, 4]
```

Explanation:

- **Time Complexity**: O(n), where `n` is the length of the array. We are slicing the array and concatenating, both of which take linear time.
- **Space Complexity**: O(1), as we modify the array in-place without using extra space. We just update the array itself.

---

1. Missing and Repeating in Array

This problem requires finding the missing and repeating elements in an array.

Best Approach: **O(n)** time complexity with **O(1)** space complexity using the mathematical properties of sum and sum of squares.

```
python
CopyEdit
def find_missing_and_repeating(arr):
    n = len(arr)
    sum_n = n * (n + 1) // 2  # Sum of first n natural number
s
    sum_n_square = (n * (n + 1) * (2 * n + 1)) // 6  # Sum of
squares of first n natural numbers

    sum_arr = sum(arr)  # Sum of elements in the array
    sum_arr_square = sum(x**2 for x in arr)  # Sum of squares
of elements in the array

    # Find the difference between the sum of numbers and the
sum of the array
    diff_sum = sum_n - sum_arr  # Missing - Repeating
    diff_square = sum_n_square - sum_arr_square  # Missing^2
- Repeating^2

    # Solving the system of equations
    diff_square -= diff_sum**2
    repeating = (diff_sum + diff_square // diff_sum) // 2
    missing = repeating - diff_sum
```

```
    return missing, repeating

# Example
arr = [4, 3, 2, 7, 8, 2, 1]
print(find_missing_and_repeating(arr))  # Output: (5, 2)
```

Explanation:

- **Time Complexity**: O(n), where `n` is the length of the array. We calculate the sum and sum of squares in linear time.
- **Space Complexity**: O(1), as we are using constant extra space for variables.

---

2. Remove All Occurrences of an Element

This problem requires removing all occurrences of a specified element from an array.

Best Approach: **O(n)** time complexity with **O(1)** space complexity by using the two-pointer technique.

```
python
CopyEdit
def remove_occurrences(arr, element):
    index = 0
    for i in range(len(arr)):
        if arr[i] != element:
            arr[index] = arr[i]
            index += 1
    return arr[:index]

# Example
arr = [3, 2, 2, 3, 4, 2, 5]
element = 2
print(remove_occurrences(arr, element))  # Output: [3, 3, 4,
```

```
5]
```

Explanation:

- **Time Complexity**: O(n), where `n` is the length of the array. We only iterate through the array once.
- **Space Complexity**: O(1), as we are modifying the array in place without using extra space.

---

3. Remove Duplicates from Sorted Array

This problem requires removing duplicates from a sorted array and returning the length of the array with no duplicates.

Best Approach: **O(n)** time complexity with **O(1)** space complexity.

```
python
CopyEdit
def remove_duplicates(arr):
    if len(arr) == 0:
        return 0

    unique_index = 1  # Index to store unique elements
    for i in range(1, len(arr)):
        if arr[i] != arr[i - 1]:
            arr[unique_index] = arr[i]
            unique_index += 1
    return unique_index

# Example
arr = [1, 1, 2, 2, 3, 3, 4]
length = remove_duplicates(arr)
print(arr[:length])  # Output: [1, 2, 3, 4]
```

Explanation:

- **Time Complexity**: O(n), where `n` is the length of the array. We iterate through the array only once.
- **Space Complexity**: O(1), as we modify the array in place without using extra space.

Practice Questions :

Easy Level

1. Find the Kth Smallest Element in an Array

- **Problem**: Given an array and an integer `k`, find the kth smallest element in the array.
- **Solution** (Using QuickSelect algorithm):

```
python
CopyEdit
def quickselect(arr, k):
    if len(arr) == 1:
        return arr[0]
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    if k < len(left):
        return quickselect(left, k)
```

```
    elif k < len(left) + len(middle):
        return pivot
    else:
        return quickselect(right, k - len(left) - len(midd
le))

# Example
arr = [12, 3, 5, 7, 19]
k = 2
print(quickselect(arr, k))  # Output: 5
```

- **Time Complexity**: O(n) (Average case for QuickSelect)
- **Space Complexity**: O(n)

---

2. Find All Pairs with Given Sum

- **Problem**: Given an array and a sum, find all unique pairs of integers in the array that add up to the sum.
- **Solution**:

```
python
CopyEdit
def find_pairs(arr, target_sum):
    seen = set()
    pairs = []
    for num in arr:
        complement = target_sum - num
        if complement in seen:
            pairs.append((complement, num))
        seen.add(num)
    return pairs

# Example
arr = [1, 4, 5, 3, 2, 6]
```

```
target_sum = 7
print(find_pairs(arr, target_sum))  # Output: [(4, 3), (5,
2)]
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(n)
-

---

3. Check if Array is Sorted

- **Problem**: Check if an array is sorted in non-decreasing order.
- **Solution**:

```
python
CopyEdit
def is_sorted(arr):
    return all(arr[i] <= arr[i+1] for i in range(len(arr)-
1))

# Example
arr = [1, 2, 2, 3]
print(is_sorted(arr))  # Output: True
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(1)

4. Move Zeros to the Beginning

- **Problem**: Move all zeros in an array to the beginning without changing the order of non-zero elements.

- **Solution**:

```
python
CopyEdit
def move_zeros_to_beginning(arr):
    index = len(arr) - 1
    # Traverse the array in reverse and fill non-zero element
s
    for num in reversed(arr):
        if num != 0:
            arr[index] = num
            index -= 1
    # Fill remaining positions with zeros
    for i in range(index + 1):
        arr[i] = 0
    return arr

# Example
arr = [0, 1, 0, 3, 12]
print(move_zeros_to_beginning(arr))  # Output: [0, 0, 1, 3, 1
2]
```

Medium Level

1. **Find Missing Number in Array**
    - **Problem**: Given an array of n-1 integers in the range of 1 to n, find the missing number.
    - **Solution**:

```
python
CopyEdit
def find_missing(arr, n):
    total_sum = n * (n + 1) // 2
    return total_sum - sum(arr)
```

```
# Example
arr = [1, 2, 4, 5, 6]
print(find_missing(arr, 6))  # Output: 3
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(1)

---

1. **Find the Majority Element**
   - **Problem**: Find the element that appears more than n/2 times in an array.
   - **Solution** (Boyer-Moore Voting Algorithm):

```
python
CopyEdit
def majority_element(arr):
    count = 0
    candidate = None
    for num in arr:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)
    return candidate

# Example
arr = [3, 3, 4, 2, 4, 4, 2, 4, 4]
print(majority_element(arr))  # Output: 4
```

   - **Time Complexity**: O(n)
   - **Space Complexity**: O(1)

---

1. **Find the Pair with Given Sum**

- **Problem**: Given an array and a sum, find two numbers that add up to the sum.
- **Solution**:

```
python
CopyEdit
def find_pair(arr, target_sum):
    seen = set()
    for num in arr:
        if target_sum - num in seen:
            return True
        seen.add(num)
    return False

# Example
arr = [10, 15, 3, 7]
target_sum = 17
print(find_pair(arr, target_sum))  # Output: True
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(n)

Hard Level

1. **Rotate the Array by K Steps**
   - **Problem**: Rotate the array to the right by `k` steps.
   - **Solution**:

```
python
CopyEdit
def rotate_array(arr, k):
    k = k % len(arr)
    return arr[-k:] + arr[:-k]
```

```
# Example
arr = [1, 2, 3, 4, 5, 6, 7]
k = 3
print(rotate_array(arr, k))  # Output: [5, 6, 7, 1, 2, 3, 4]
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(n)

---

1. **Find the Longest Consecutive Sequence**
   - **Problem**: Given an unsorted array, find the longest consecutive sequence of integers.
   - **Solution** (Using a Set):

```
python
CopyEdit
def longest_consecutive(arr):
    nums = set(arr)
    longest = 0

    for num in arr:
        if num - 1 not in nums:
            current_num = num
            current_streak = 1

            while current_num + 1 in nums:
                current_num += 1
                current_streak += 1

            longest = max(longest, current_streak)

    return longest
```

```
# Example
arr = [100, 4, 200, 1, 3, 2]
print(longest_consecutive(arr))  # Output: 4
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(n)

---

1. **Subarray with Given Sum**

- **Problem**: Find a subarray with a given sum (if it exists).
- **Solution** (Using Sliding Window):

```
python
CopyEdit
def subarray_sum(arr, target_sum):
    current_sum = 0
    start = 0

    for end in range(len(arr)):
        current_sum += arr[end]

        while current_sum > target_sum and start <= end:
            current_sum -= arr[start]
            start += 1

        if current_sum == target_sum:
            return arr[start:end + 1]

    return None

# Example
arr = [1, 4, 20, 3, 10, 5]
target_sum = 33
```

```
print(subarray_sum(arr, target_sum))  # Output: [20, 3, 1
0]
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(1)