Day 1

Learn the basics of arrays: indexing, access, and memory allocation.

=> Understanding the basics of arrays in JavaScript is crucial, especially regarding indexing, access, and memory allocation. Here's a detailed breakdown:

1. **Array Indexing**

- Arrays in JavaScript are **zero-indexed**, which means the first element is at index `0`, the second element is at index `1`, and so on.
- You can access elements of an array using the bracket `[]` notation and providing the index number inside the brackets.

Example:

```
const fruits = ['apple', 'banana', 'cherry'];

// Accessing elements by index
console.log(fruits[0]); // Output: 'apple'
console.log(fruits[1]); // Output: 'banana'
console.log(fruits[2]); // Output: 'cherry'
```

2. **Array Access**

Accessing elements in a JavaScript array is very efficient. It's an **O(1)** (constant time) operation, meaning that no matter how large the array is, accessing an element by index always takes the same amount of time.

- **Accessing an element:** You retrieve an element from the array by its index.
- **Modifying an element:** You can also update an element at a specific index using the same bracket notation.

Example:

```
const fruits = ['apple', 'banana', 'cherry'];

// Access element
let firstFruit = fruits[0];  // 'apple'

// Modify element
fruits[1] = 'blueberry';  // Changes 'banana' to 'blueberry'
console.log(fruits);  // Output: ['apple', 'blueberry', 'cherry']
```

3. **Memory Allocation for Arrays**

Unlike in lower-level languages (such as C), JavaScript arrays are **dynamic** and can grow or shrink in size. JavaScript arrays are more like **list-like objects** and are not stored contiguously in memory. Instead, the JavaScript engine manages memory allocation dynamically, allowing flexibility but with some performance trade-offs.

- **Dynamic Arrays:** JavaScript arrays can hold different types of values (numbers, strings, objects, etc.), and their size can change dynamically. The memory allocation is **non-contiguous**—not like static arrays in other languages (e.g., C or Java), where arrays occupy consecutive memory blocks.
- **Internal Mechanism:** While JavaScript abstracts away memory management from the developer, under the hood, JavaScript engines optimize how arrays are stored. Some engines store arrays as hash tables or arrays of pointers when the elements are sparse.

Example:

```
const mixedArray = [1, 'apple', { name: 'banana' }, [2, 3]];
console.log(mixedArray);
// Output: [1, 'apple', { name: 'banana' }, [2, 3]]

// You can even add more elements dynamically
mixedArray.push('new element');
console.log(mixedArray);
// Output: [1, 'apple', { name: 'banana' }, [2, 3], 'new element']
```

4. **Array Length**

- The `length` property of an array gives the number of elements in the array. It is **dynamic** and automatically adjusts when you add or remove elements.

Example:

```
const fruits = ['apple', 'banana', 'cherry'];
console.log(fruits.length); // Output: 3

fruits.push('orange');
console.log(fruits.length); // Output: 4
```

5. **Sparse Arrays**

JavaScript arrays can also be **sparse**, meaning they can have empty or undefined indices. A sparse array doesn't hold elements for every index.

Example:

```
const sparseArray = [];
sparseArray[5] = 'apple';
console.log(sparseArray);  // Output: [ <5 empty items>, 'apple' ]
console.log(sparseArray.length);  // Output: 6 (length includes empty slots)
```

Memory Allocation in Detail

JavaScript arrays behave differently than typical arrays in low-level languages:

- **Heap Memory**: Arrays and objects in JavaScript are stored in the heap, a memory space for dynamic allocations.
- **Memory Fragmentation**: Since JavaScript arrays can store different types of data (strings, numbers, objects), and their size is dynamic, memory fragmentation may occur where items are not stored consecutively.

Important Points:

- Arrays in JavaScript are **dynamic and flexible**, unlike static arrays in lower-level languages.
- They are **non-typed** (can store different data types).
- Memory is dynamically allocated and managed by the JavaScript engine.

Summary:

- **Indexing**: JavaScript arrays are zero-indexed.
- **Access**: Fast, O(1) access using the index.
- **Memory**: JavaScript arrays are dynamic and managed by the JavaScript engine. They are not stored contiguously in memory, unlike arrays in languages like C.
- **Length**: The `length` property dynamically changes as elements are added or removed.

With these fundamentals, you'll have a strong grasp of how arrays work in JavaScript from both an access and memory management perspective!

Study the advantages and limitations of arrays.

=> Advantages and Limitations of Arrays in JavaScript

Arrays are an essential data structure in JavaScript, but like any tool, they come with both advantages and limitations. Understanding these can help you determine when to use arrays and when other data structures might be more appropriate.

Advantages of Arrays in JavaScript

1. **Ease of Use**

- **Built-in Functions**: JavaScript arrays come with many built-in methods like `.map()`, `.filter()`, `.reduce()`, `.forEach()`, `.push()`, and many more. These methods make it easy to manipulate arrays without the need for manual loops or complicated logic.
- **Dynamic Size**: JavaScript arrays can grow and shrink dynamically. You don't need to pre-define the size of an array like in some other programming languages (e.g., C, C++). This makes arrays highly flexible when dealing with unknown amounts of data.

Example:

```
const arr = [];
arr.push(1);  // [1]
arr.push(2);  // [1, 2]
arr.pop();    // [1]
```

2. **Efficient Access (O(1) Time Complexity)**

- Arrays provide **constant-time access** to elements by their index. If you know the index of an element, you can retrieve or update it efficiently.

Example:

```
const arr = [10, 20, 30, 40];
console.log(arr[2]);  // Output: 30 (O(1) access)
```

3. **Ordered Structure**

- Arrays maintain the order of elements. When you add or remove elements, they remain in sequence, which is useful when order is important (e.g., in lists of names, tasks, etc.).

4. **Flexible Data Types**

- Arrays in JavaScript can store elements of **different data types** (e.g., numbers, strings, objects, other arrays). This flexibility can simplify certain tasks that require handling heterogeneous data.

Example:

```
const mixedArray = [1, "apple", true, { id: 1, name: "John" }];
```

5. **Array-Like Methods and Iterability**

- Arrays are **iterable**, making it easy to use `for...of` loops, and they support various array-like operations (such as `map`, `reduce`, and `filter`). This provides a clean, functional approach to handling array elements.

Example:

```
const numbers = [1, 2, 3];
const squared = numbers.map(num => num * 2);
console.log(squared);  // Output: [2, 4, 6]
```

6. **Memory Efficiency**

- JavaScript arrays are **memory-efficient** for small datasets. They dynamically manage their memory allocations, which means you don’t have to worry about memory overhead like you do in lower-level languages.

Limitations of Arrays in JavaScript

1. **Non-Contiguous Memory Allocation**

- Unlike arrays in languages like C or Java, JavaScript arrays do not guarantee **contiguous memory allocation**. This makes accessing large datasets potentially slower than in languages with static arrays.

Example:

In C/C++, arrays are stored in contiguous memory blocks, ensuring that the next element is just one memory slot away. In JavaScript, due to dynamic sizing and different data types, arrays are internally optimized by the JavaScript engine, and memory can be fragmented.

2. **Performance on Insertions/Deletions**

- **Insertion and deletion in the middle or beginning** of an array can be costly because all subsequent elements must be shifted to accommodate the change.
  - **Insertion at the end** (`.push()`) is efficient because no shifting is required.
  - **Deletion from the end** (`.pop()`) is also efficient.

Example:

```
const arr = [1, 2, 3, 4];
arr.splice(1, 0, 5);  // Insertion at index 1, costly due to shifting
// Output: [1, 5, 2, 3, 4]
```

- **Time complexity**:
  - Insertion at the beginning: O(n)
  - Deletion at the beginning: O(n)
  - Insertion at the end: O(1)
  - Deletion at the end: O(1)

3. **Sparse Arrays**

- JavaScript arrays can be **sparse** (meaning they can have empty or undefined elements), but this can lead to **unexpected results** and performance issues when looping or applying array methods.

Example:

```
const arr = [];
arr[10] = "apple";  // Sparse array
console.log(arr.length);  // Output: 11
```

4. **Lack of Type Safety**

- Arrays in JavaScript can store **heterogeneous data types**, which can lead to potential bugs or misuse. You cannot enforce the type of elements stored in the array, which can cause unexpected behavior in complex codebases.

Example:

```
const mixedArray = [1, "two", { three: 3 }];
```

- This flexibility can also make the code harder to debug when it’s not clear what kind of data is being processed.

5. **No Direct Support for Multi-dimensional Arrays**

- JavaScript doesn’t have built-in support for **multi-dimensional arrays**. To create them, you need to use arrays of arrays, which can be cumbersome to manage, especially when working with large datasets like matrices.

Example:

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log(matrix[1][1]);  // Output: 5
```

6. **Overhead for Non-numeric Indexes**

- Arrays in JavaScript can store non-numeric properties (like an object), but this adds overhead because the array will behave more like an object rather than an indexed collection.
- **Array-like objects** (with non-numeric indexes) lose the advantages of traditional arrays like fast access.

Example:

```
const arr = [1, 2, 3];
arr['name'] = 'myArray';  // Adds a non-numeric property
console.log(arr.name);    // Output: 'myArray'
```

7. **Fixed Indexing Limits**

- JavaScript arrays have a **maximum length** depending on the JavaScript engine (such as V8 in Chrome), typically around **2^32 - 1** (approximately 4.29 billion elements). While this is usually more than sufficient, it's still a limit for extremely large data sets.

Summary

Advantages	Limitations
Dynamic size and flexibility	Non-contiguous memory allocation
Built-in methods for ease of use	Costly insertion/deletion in the middle/beginning
Constant-time element access (O(1))	Lack of type safety (heterogeneous elements)
Can hold mixed data types	Sparse arrays can cause unexpected results
Maintains the order of elements	No direct support for multi-dimensional arrays
Memory efficiency for small datasets	Overhead for non-numeric indexes

Conclusion:

Arrays in JavaScript are powerful, flexible, and easy to use due to their dynamic nature and built-in methods. However, they come with limitations like memory fragmentation, performance hits on insertion/deletion at the beginning or middle, and lack of strict type enforcement. Understanding these trade-offs will help you make informed decisions about when to use arrays and when to consider alternative data structures (like objects, maps, sets, or linked lists).

Day 2 : Array Operations and Complexity

Explore common array operations: insertion, deletion, searching.

=> In JavaScript, arrays are versatile data structures, and common operations like **insertion**, **deletion**, and **searching** are frequently used in various programming tasks. Let's explore each of these operations with examples and explanations.

---

1. Insertion in Arrays

Insertion refers to adding elements to an array. There are several ways to insert elements into an array, depending on where you want to add the element (beginning, middle, or end).

At the End of the Array

- **Method**: Use the `.push()` method.
- **Time Complexity**: O(1)

```
const arr = [1, 2, 3];
arr.push(4);  // Insert 4 at the end
console.log(arr);  // Output: [1, 2, 3, 4]
```

At the Beginning of the Array

- **Method**: Use the `.unshift()` method.
- **Time Complexity**: O(n) – All elements must be shifted to the right.

```
const arr = [1, 2, 3];
arr.unshift(0);  // Insert 0 at the beginning
console.log(arr);  // Output: [0, 1, 2, 3]
```

In the Middle of the Array

- **Method**: Use the `.splice()` method to insert elements at a specific index.
- **Time Complexity**: O(n) – Elements after the insertion point need to be shifted.

```
const arr = [1, 2, 4];
arr.splice(2, 0, 3);  // Insert 3 at index 2
console.log(arr);  // Output: [1, 2, 3, 4]
```

---

2. Deletion from Arrays

Deletion refers to removing elements from an array. Like insertion, deletion can occur at the beginning, middle, or end of an array.

From the End of the Array

- **Method**: Use the `.pop()` method.
- **Time Complexity**: O(1)

```
const arr = [1, 2, 3, 4];
arr.pop();  // Remove the last element
console.log(arr);  // Output: [1, 2, 3]
```

From the Beginning of the Array

- **Method**: Use the `.shift()` method.
- **Time Complexity**: O(n) – All elements must be shifted to the left.

```
const arr = [1, 2, 3, 4];
arr.shift();  // Remove the first element
console.log(arr);  // Output: [2, 3, 4]
```

From the Middle of the Array

- **Method**: Use the `.splice()` method to remove elements at a specific index.
- **Time Complexity**: O(n) – Elements after the deletion point need to be shifted.

```
const arr = [1, 2, 3, 4];
arr.splice(2, 1);  // Remove 1 element at index 2 (removes 3)
console.log(arr);  // Output: [1, 2, 4]
```

---

3. Searching in Arrays

Searching is the process of finding an element in an array. JavaScript provides multiple ways to search for an element.

Using `indexOf()`

- Finds the first occurrence of an element in the array and returns its index. Returns `-1` if not found.
- **Time Complexity**: O(n)

```
const arr = [1, 2, 3, 4];
console.log(arr.indexOf(3));  // Output: 2
console.log(arr.indexOf(5));  // Output: -1
```

Using `includes()`

- Checks if an array contains a specific element. Returns `true` or `false`.
- **Time Complexity**: O(n)

```
const arr = [1, 2, 3, 4];
console.log(arr.includes(3));  // Output: true
console.log(arr.includes(5));  // Output: false
```

Using `find()`

- Returns the **first element** that satisfies the provided condition (predicate function). If no such element exists, it returns `undefined`.
- **Time Complexity**: O(n)

```
const arr = [1, 2, 3, 4];
const found = arr.find(element => element > 2);
console.log(found);  // Output: 3
```

Using `findIndex()`

- Returns the **index** of the first element that satisfies the provided condition. Returns `-1` if no such element exists.
- **Time Complexity**: O(n)

```
const arr = [1, 2, 3, 4];
const foundIndex = arr.findIndex(element => element > 2);
console.log(foundIndex);  // Output: 2
```

Using `filter()`

- Returns a new array containing all elements that satisfy the provided condition.
- **Time Complexity**: O(n)

```
const arr = [1, 2, 3, 4];
const filtered = arr.filter(element => element > 2);
console.log(filtered);  // Output: [3, 4]
```

Summary of Array Operations

Operation	Method	Description	Time Complexity
Insertion at the End	`.push()`	Adds an element to the end of the array	O(1)
Insertion at the Beginning	`.unshift()`	Adds an element to the start of the array	O(n)
Insertion in the Middle	`.splice()`	Inserts an element at a specified index	O(n)
Deletion from the End	`.pop()`	Removes the last element	O(1)
Deletion from the Beginning	`.shift()`	Removes the first element	O(n)
Deletion from the Middle	`.splice()`	Removes an element at a specified index	O(n)
Search by Value	`.indexOf()`	Returns the index of the element if found, otherwise -1	O(n)
Check if Value Exists	`.includes()`	Returns `true` if element exists, otherwise `false`	O(n)
Find First Match	`.find()`	Returns the first element matching the condition	O(n)
Find Index of First Match	`.findIndex()`	Returns the index of the first element matching the condition	O(n)
Filter by Condition	`.filter()`	Returns a new array with all elements that match the condition	O(n)

Understanding these array operations, their usage, and time complexities will help you write more efficient and readable JavaScript code.

Understand time and space complexity analysis for array operations.

=> Time and space complexity analysis is essential for understanding the efficiency of array operations in JavaScript. Let's break down the time and space complexity of common array operations like **accessing**, **inserting**, **deleting**, and **searching**.

1. Time Complexity Analysis of Array Operations

Time complexity describes how the runtime of an algorithm or operation grows as the input size increases. In the case of arrays, time complexity depends on where in the array the operation is being performed (beginning, middle, or end).

Accessing Elements

- **Access by Index**: Accessing an element by index in an array is a constant-time operation because arrays in JavaScript are **contiguous** in memory, meaning each element can be accessed directly.
- **Time Complexity**: **O(1)**

```
const arr = [10, 20, 30];
console.log(arr[1]);  // Output: 20, accessed in O(1) time
```

Inserting Elements

- **At the End** (`.push()`): Inserting at the end of an array is efficient since no elements need to be shifted.
    - **Time Complexity**: **O(1)**

```
const arr = [10, 20, 30];
arr.push(40);  // Output: [10, 20, 30, 40], done in O(1) time
```

- **At the Beginning** (`.unshift()`): Inserting at the beginning requires shifting all existing elements to the right, which takes time proportional to the size of the array.
    - **Time Complexity**: **O(n)**

```
const arr = [10, 20, 30];
arr.unshift(5);  // Output: [5, 10, 20, 30], done in O(n) time
```

- **In the Middle** (`.splice()`): Inserting an element at a specific index involves shifting elements to the right, and it takes time proportional to the number of elements after the insertion point.
    - **Time Complexity**: **O(n)**

```
const arr = [10, 20, 30];
arr.splice(1, 0, 15);  // Insert 15 at index 1, output: [10, 15, 20, 30]
```

Deleting Elements

- **From the End** (`.pop()`): Deleting the last element is efficient as no shifting is required.
    - **Time Complexity**: **O(1)**

```
const arr = [10, 20, 30];
arr.pop();  // Output: [10, 20], done in O(1) time
```

- **From the Beginning** (`.shift()`): Deleting from the beginning requires shifting all elements to the left to fill the gap, making it slower.
    - **Time Complexity**: **O(n)**

```
const arr = [10, 20, 30];
arr.shift();  // Output: [20, 30], done in O(n) time
```

- **From the Middle** (`.splice()`): Deleting an element in the middle requires shifting elements to the left, proportional to the number of elements after the deletion point.
    - **Time Complexity**: **O(n)**

```
const arr = [10, 20, 30];
arr.splice(1, 1);  // Remove element at index 1, output: [10, 30]
```

Searching Elements

- **Linear Search** (`.indexOf()`, `.find()`, `.includes()`): Searching for an element in an array typically requires iterating through the array to find the element. This makes it a linear-time operation.
    - **Time Complexity**: **O(n)**

```
const arr = [10, 20, 30, 40];
console.log(arr.indexOf(30));  // Output: 2, found in O(n) time
```

---

2. Space Complexity Analysis of Array Operations

Space complexity refers to the amount of memory an operation uses relative to the size of the input.

- **Access by Index**: Accessing an element by its index does not require additional space, as it simply retrieves the value from the existing array.
    - **Space Complexity**: **O(1)**
- **Insertion and Deletion**: Inserting or deleting elements in arrays involves temporary memory for shifting operations, but the space required for shifting is proportional to the array size.
    - **Space Complexity**: **O(1)** for the operation itself (the array size remains the same).
    - If creating a new array to store the shifted elements, the space complexity becomes **O(n)**.
- **Searching**: Searching does not require additional space unless results are stored in a new array.
    - **Space Complexity**: **O(1)** for operations like `.find()` or `.includes()` that return a single value.
    - **O(n)** if storing results, as in the case of `.filter()`.

Summary Table of Time and Space Complexity

Operation	Time Complexity	Space Complexity
Access by Index	O(1)	O(1)
Insert at the End	O(1)	O(1)
Insert at the Beginning	O(n)	O(1)
Insert in the Middle	O(n)	O(1)
Delete from the End	O(1)	O(1)
Delete from the Beginning	O(n)	O(1)
Delete from the Middle	O(n)	O(1)
Search (Linear)	O(n)	O(1)

Additional Considerations

- **Dynamic Arrays**: JavaScript arrays are dynamic, meaning they can grow or shrink in size. Internally, when the array grows beyond its current capacity, JavaScript might reallocate memory and copy the elements to a new location. This **reallocation** is usually an amortized **O(1)** operation but can incur occasional overhead when expanding the array.
- **Sparse Arrays**: JavaScript allows sparse arrays (arrays with empty or undefined slots), which can affect performance. Accessing an element in a sparse array may involve searching through the gaps, potentially making operations slower in practice.

Conclusion

The **time complexity** of array operations primarily depends on the position in the array where the operation occurs. Accessing elements is fast, while insertion and deletion at the beginning or middle are slower due to shifting elements. Searching takes linear time since arrays don't have an efficient way to search without iterating through elements. **Space complexity** is generally low, but creating new arrays or shifting elements increases memory usage.

Understanding the time and space complexities of these operations is crucial for writing efficient code, especially when dealing with large datasets.

Day 3 : Array Traversal and Searching

Learn various methods for array traversal: linear, reverse, and 2D arrays.

=> Array traversal refers to the process of visiting each element in an array, either to perform some operation on each element or to access the values. Let's explore different ways to traverse arrays in JavaScript, including **linear traversal**, **reverse traversal**, and **2D array traversal**.

---

1. Linear Traversal

Linear traversal is the most common way of accessing each element of a one-dimensional array from the beginning to the end. Here are a few ways to perform linear traversal in JavaScript.

Using `for` Loop:

```
const arr = [10, 20, 30, 40, 50];

for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
// Output: 10 20 30 40 50
```

Using `forEach()` Method:

The `forEach()` method is specifically designed for iterating through arrays.

```
arr.forEach(function(element) {
    console.log(element);
});
// Output: 10 20 30 40 50
```

Using `for...of` Loop:

The `for...of` loop is a modern way to iterate through arrays without accessing the index.

```
for (const element of arr) {
    console.log(element);
}
// Output: 10 20 30 40 50
```

---

2. Reverse Traversal

Reverse traversal involves visiting the elements of an array from the last index to the first.

Using `for` Loop in Reverse:

You can reverse the direction of the `for` loop to start from the last element and move towards the first.

```
const arr = [10, 20, 30, 40, 50];

for (let i = arr.length - 1; i >= 0; i--) {
    console.log(arr[i]);
}
// Output: 50 40 30 20 10
```

Using `reverse()` Method:

The `reverse()` method can be used to reverse the array in place, followed by a regular traversal.

```
arr.reverse().forEach(function(element) {
    console.log(element);
});
// Output: 50 40 30 20 10
```

However, note that `reverse()` **modifies the original array**. If you want to keep the array unchanged, you can use a copy:

```
[...arr].reverse().forEach(element => console.log(element));
// Output: 50 40 30 20 10
```

---

3. 2D Array Traversal

A two-dimensional (2D) array can be visualized as an array of arrays, often used to represent matrices. You can traverse it using nested loops.

Creating a 2D Array:

```
const matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

Row-wise Traversal (Left to Right):

To access elements row by row:

```
for (let i = 0; i < matrix.length; i++) {
    for (let j = 0; j < matrix[i].length; j++) {
        console.log(matrix[i][j]);
    }
}
// Output: 1 2 3 4 5 6 7 8 9
```

Column-wise Traversal (Top to Bottom):

You can also traverse column by column by switching the order of loops:

```
for (let j = 0; j < matrix[0].length; j++) {
    for (let i = 0; i < matrix.length; i++) {
        console.log(matrix[i][j]);
    }
}
// Output: 1 4 7 2 5 8 3 6 9
```

Diagonal Traversal:

To traverse diagonally (from top-left to bottom-right):

```
for (let i = 0; i < matrix.length; i++) {
    console.log(matrix[i][i]);
}
// Output: 1 5 9
```

Reverse Diagonal Traversal (Bottom-left to Top-right):

```
for (let i = 0; i < matrix.length; i++) {
    console.log(matrix[i][matrix.length - 1 - i]);
}
// Output: 3 5 7
```

Summary of Traversal Methods:

- **Linear Traversal**: Access each element of a 1D array from start to end.
- **Reverse Traversal**: Access elements of a 1D array from end to start.
- **2D Array Traversal**: Traverse a 2D array row-wise, column-wise, or diagonally using nested loops.

These traversal techniques form the foundation for working with arrays and matrices in JavaScript. You can extend these ideas for more complex operations like searching, sorting, and matrix manipulation.

Practice searching algorithms such as linear search and binary search.

=> Searching Algorithms: Linear Search and Binary Search

Searching algorithms are crucial when you need to find an element in a list or array. We'll cover **Linear Search** and **Binary Search** in JavaScript, which are widely used for searching in arrays.

---

1. Linear Search

Linear Search is a straightforward algorithm that checks each element of the array sequentially until the target element is found or the end of the array is reached. It works for **unsorted** arrays.

Algorithm Steps:

1. Start from the first element of the array.
2. Compare the target value with each element of the array.
3. If the target is found, return the index.
4. If the target is not found by the end of the array, return `-1`.

Time Complexity:

- **Best Case**: O(1) (if the target is the first element).
- **Average Case**: O(n).
- **Worst Case**: O(n) (if the target is the last element or not present).

Code Example:

```
function linearSearch(arr, target) {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === target) {
            return i;  // Target found, return index
        }
    }
    return -1;  // Target not found
}

// Test the function
const arr = [10, 20, 30, 40, 50];
const target = 30;
const index = linearSearch(arr, target);

console.log(index);  // Output: 2 (since 30 is at index 2)
```

---

2. Binary Search

Binary Search is an efficient algorithm that works only on **sorted** arrays. It repeatedly divides the search space in half until the target element is found or the search space is empty.

Algorithm Steps:

1. Find the middle element of the array.
2. Compare the middle element with the target.
    - If they are equal, return the middle index.
    - If the target is smaller, search the left half.
    - If the target is larger, search the right half.
3. Repeat the process until the target is found or the search space becomes empty.

Time Complexity:

- **Best Case**: O(1).
- **Average Case**: O(log n).
- **Worst Case**: O(log n).

Code Example:

```
function binarySearch(arr, target) {
    let left = 0;
    let right = arr.length - 1;

    while (left <= right) {
        const middle = Math.floor((left + right) / 2);

        if (arr[middle] === target) {
            return middle;  // Target found
        } else if (arr[middle] < target) {
            left = middle + 1;  // Search the right half
        } else {
            right = middle - 1;  // Search the left half
        }
    }

    return -1;  // Target not found
}

// Test the function
const sortedArr = [10, 20, 30, 40, 50];
const targetVal = 40;
const result = binarySearch(sortedArr, targetVal);

console.log(result);  // Output: 3 (since 40 is at index 3)
```

Comparison: Linear Search vs. Binary Search

Characteristic	Linear Search	Binary Search
Time Complexity	O(n)	O(log n)
Works on Unsorted Arrays	Yes	No (requires sorted array)
Best Case Performance	O(1)	O(1)
Space Complexity	O(1)	O(1)
Usage	Small or unsorted arrays	Large sorted arrays

Practice Exercises:

1. **Linear Search**:
    - Try modifying the linear search algorithm to count the number of comparisons made.
    - Implement linear search on an array of objects.
2. **Binary Search**:
    - Implement binary search recursively.
    - Modify the binary search to work on arrays sorted in descending order.

These two algorithms cover a wide range of use cases for searching in arrays, and their time complexity characteristics help you decide which one to use based on the data.

Day 4 : Sorting Algorithms

Study sorting algorithms applicable to arrays: bubble sort, selection sort, insertion sort.

=> Sorting algorithms are essential for organizing data efficiently. Let's look at three simple and commonly used sorting algorithms: **Bubble Sort**, **Selection Sort**, and **Insertion Sort**. Each algorithm operates on arrays in JavaScript.

---

1. Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the array is sorted.

Algorithm Steps:

1. Compare adjacent elements in the array.
2. Swap them if they are in the wrong order (i.e., larger element comes first).
3. Repeat the process for all elements until the array is sorted.

Time Complexity:

- **Best Case**: O(n) (when the array is already sorted).
- **Average Case**: O(n^2).
- **Worst Case**: O(n^2).

Code Example:

```
function bubbleSort(arr) {
    let n = arr.length;
    let swapped;

    do {
        swapped = false;
        for (let i = 0; i < n - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                // Swap elements
                let temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
                swapped = true;
            }
        }
        n--;  // Reduce the range for the next pass
    } while (swapped);

    return arr;
}

// Test the function
let arr = [64, 34, 25, 12, 22, 11, 90];
console.log(bubbleSort(arr));  // Output: [11, 12, 22, 25, 34, 64, 90]
```

---

2. Selection Sort

Selection Sort works by dividing the array into a sorted and an unsorted part. The smallest (or largest) element is selected from the unsorted part and swapped with the leftmost unsorted element, moving the boundary between sorted and unsorted parts.

Algorithm Steps:

1. Find the smallest element in the unsorted part of the array.
2. Swap it with the first element of the unsorted part.
3. Move the boundary between sorted and unsorted parts and repeat.

Time Complexity:

- **Best Case**: O(n²).
- **Average Case**: O(n²).
- **Worst Case**: O(n²).

Code Example:

```
function selectionSort(arr) {
    let n = arr.length;

    for (let i = 0; i < n - 1; i++) {
        // Assume the first element of the unsorted part is the minimum
        let minIndex = i;

        for (let j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;  // Update the minIndex if a smaller element is found
            }
        }

        // Swap the minimum element with the first unsorted element
        if (minIndex !== i) {
            let temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }

    return arr;
}

// Test the function
let arr = [64, 25, 12, 22, 11];
console.log(selectionSort(arr));  // Output: [11, 12, 22, 25, 64]
```

---

3. Insertion Sort

Insertion Sort builds the sorted array one element at a time. It works similarly to how you might sort playing cards in your hand. It picks an element and places it in its correct position by comparing it with all elements in the sorted portion of the array.

Algorithm Steps:

1. Start from the second element, and compare it with the elements before it.
2. Insert it into its correct position in the sorted portion of the array.
3. Repeat for each subsequent element until the array is sorted.

Time Complexity:

- **Best Case**: O(n) (when the array is already sorted).
- **Average Case**: O(n²).
- **Worst Case**: O(n²).

Code Example:

```
function insertionSort(arr) {
    for (let i = 1; i < arr.length; i++) {
        let key = arr[i];  // The current element to be inserted
        let j = i - 1;

        // Shift elements of the sorted part that are greater than the key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        // Insert the current element into its correct position
        arr[j + 1] = key;
    }

    return arr;
}

// Test the function
let arr = [12, 11, 13, 5, 6];
console.log(insertionSort(arr));  // Output: [5, 6, 11, 12, 13]
```

Comparison of Sorting Algorithms:

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Space Complexity	Stable?
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Yes
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	No
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Yes

- **Bubble Sort**: Best for nearly sorted data, easy to implement but inefficient for large datasets.
- **Selection Sort**: Simple but inefficient. It performs the same number of comparisons regardless of the data's initial order.
- **Insertion Sort**: Efficient for small datasets or partially sorted arrays, with better performance than Bubble and Selection Sort in the best case.

Practice Exercises:

1. **Bubble Sort**:
    - Modify the bubble sort to sort in descending order.
    - Implement an optimized bubble sort that terminates early if no swaps are made during a pass.
2. **Selection Sort**:
    - Modify the selection sort to sort in descending order.
3. **Insertion Sort**:
    - Modify the insertion sort to handle sorting of arrays with duplicate values efficiently.

These algorithms help you get started with sorting, but there are more advanced algorithms like **Merge Sort** and **Quick Sort** that are more efficient for larger datasets.

Analyze time complexity and compare these algorithms.

=> Time Complexity Analysis of Sorting Algorithms

Time complexity is a crucial metric for evaluating the efficiency of algorithms, particularly as the input size grows. Let's analyze the time complexity for **Bubble Sort**, **Selection Sort**, and **Insertion Sort** in detail and compare their performances.

---

1. Bubble Sort

Time Complexity:

- **Best Case**: O(n)If the array is already sorted, the algorithm will pass through the array once, and no swaps will be made. Thus, it terminates after one pass.
- **Average Case**: O(n²)In the average case, Bubble Sort compares and swaps elements in multiple passes, making approximately `n(n-1)/2` comparisons, leading to quadratic complexity.
- **Worst Case**: O(n²) The worst case occurs when the array is sorted in reverse order, requiring the maximum number of swaps, i.e., n passes and `n(n-1)/2` comparisons.

Space Complexity: O(1)

Bubble Sort is an **in-place sorting algorithm**, so it uses a constant amount of extra memory (no additional data structures).

---

2. Selection Sort

Time Complexity:

- **Best Case**: O(n²)Selection Sort makes no assumptions about the initial order of the array. It always performs `n(n-1)/2` comparisons, regardless of whether the array is already sorted.
- **Average Case**: O(n²)The average number of comparisons and swaps remains the same for random input, i.e., quadratic time complexity.
- **Worst Case**: O(n²) Even in the worst case, Selection Sort performs the same number of comparisons as in the average case, so its time complexity remains O(n²).

Space Complexity: O(1)

Like Bubble Sort, Selection Sort is an in-place algorithm, so it has constant space complexity.

---

3. Insertion Sort

Time Complexity:

- **Best Case**: O(n)If the array is already sorted, Insertion Sort will only make `n-1` comparisons (no shifts), so the time complexity in this case is linear, O(n).
- **Average Case**: O(n²)In the average case, Insertion Sort will perform approximately `n²/4` comparisons and shifts, leading to quadratic complexity.

- **Worst Case**: O(n²) In the worst case (when the array is sorted in reverse order), each element needs to be compared and shifted all the way to the beginning of the array, requiring `n(n-1)/2` comparisons and shifts.

Space Complexity: O(1)

Insertion Sort is also in-place, requiring no additional space.

Comparison Table

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Space Complexity	Stable?
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Yes
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	No
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Yes

Analysis and Comparison

1. **Best Case**:
    - **Insertion Sort** outperforms the other algorithms in the best case with O(n) time complexity, making it efficient for nearly sorted arrays.
    - **Bubble Sort** also has a best-case time of O(n), but it is less efficient than Insertion Sort because Bubble Sort makes unnecessary passes after the array is sorted.
    - **Selection Sort** has the same time complexity O(n²) in all cases, making it inefficient even when the array is already sorted.
2. **Average Case**:
    - All three algorithms perform similarly in the average case, with time complexity O(n²). However, **Insertion Sort** may still perform better on smaller datasets or partially sorted data due to fewer comparisons and shifts.
3. **Worst Case**:
    - The worst-case performance for all three algorithms is O(n²), but **Insertion Sort** can be better optimized by reducing unnecessary comparisons and shifts in practice.
4. **Space Complexity**:
    - All three algorithms use constant space (O(1)), making them in-place algorithms. This is advantageous in memory-constrained environments.
5. **Stability**:
    - **Insertion Sort** and **Bubble Sort** are stable sorting algorithms, meaning they maintain the relative order of elements with equal keys. **Selection Sort** is not stable because it might change the relative order of equal elements during swaps.

When to Use Each Algorithm

- **Bubble Sort**:
    - Suitable for educational purposes or small arrays. It is easy to implement but generally not used in practice due to its inefficiency with larger datasets.
- **Selection Sort**:

- Simple to implement and doesn't require many swaps, but it performs poorly in terms of comparisons. It's used in cases where writing to memory is costly (minimizes swaps).

- **Insertion Sort**:

  - Ideal for small arrays or nearly sorted arrays. It is used in hybrid algorithms like **Timsort**, which leverages Insertion Sort for small data sizes.

Conclusion

While **Bubble Sort**, **Selection Sort**, and **Insertion Sort** are easy to understand and implement, they are inefficient for large datasets due to their O(n²) time complexity in the average and worst cases. More advanced sorting algorithms, such as **Merge Sort** (O(n log n)) and **Quick Sort** (O(n log n) average case), are generally preferred in practical applications. However, **Insertion Sort** remains useful for smaller datasets and nearly sorted data due to its O(n) best-case performance.

Day 5 : Dynamic Arrays (ArrayLists)

Understand dynamic arrays (ArrayList in Java) and their advantages.

=> Dynamic Arrays (ArrayList in Java)

Dynamic arrays, like Java's `ArrayList`, provide the advantages of both arrays (for fast access) and linked lists (for flexible resizing). Unlike static arrays, whose size is fixed upon initialization, dynamic arrays can grow or shrink automatically when elements are added or removed.

Key Concepts of Dynamic Arrays (ArrayList in Java)

1. **Resizing**:

   - Unlike traditional arrays with a fixed size, dynamic arrays automatically resize when more space is needed. When the array's capacity is exhausted, it creates a larger array (often twice the current size), copies the old elements into it, and continues adding new elements.

2. **Dynamic Memory Allocation**:

   - Dynamic arrays adjust their size during runtime based on the number of elements. This flexibility is useful in scenarios where the number of elements is not known in advance.

3. **ArrayList Implementation in Java**:

   - Java provides `ArrayList`, which is a dynamic array that can grow and shrink. The size of an `ArrayList` is not fixed, and it provides methods for easy manipulation (adding, removing, searching for elements).
   - **Example**: In Java, `ArrayList` can be defined and used as:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();

        // Adding elements
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        System.out.println(numbers); // Output: [10, 20, 30]

        // Removing element
        numbers.remove(1);  // Removes the element at index 1
        System.out.println(numbers); // Output: [10, 30]
    }
}
```

Advantages of Dynamic Arrays (ArrayList)

1. **Resizing**:
   - **Flexibility**: The primary advantage of dynamic arrays is their ability to automatically resize. This removes the need to predefine the size of the array, allowing you to add as many elements as needed without worrying about capacity.
   - **Efficient Growth**: Typically, dynamic arrays like `ArrayList` double their capacity when the size limit is reached, meaning the resizing process is relatively efficient and does not occur frequently.
2. **Efficient Access (Random Access)**:
   - Just like traditional arrays, dynamic arrays provide O(1) time complexity for accessing elements by index, making them ideal for applications where you need fast random access to elements.
3. **Inserting at the End**:
   - Adding elements at the end of a dynamic array (such as using the `add()` method in Java's `ArrayList`) takes O(1) time on average. Even though resizing the array occasionally takes O(n) time, the resizing happens infrequently enough that the **amortized** time complexity of appending elements is O(1).
4. **Shrinking**:
   - Many dynamic array implementations (including Java's `ArrayList`) support shrinking, which allows the array to free up unused memory if elements are removed. This makes dynamic arrays memory efficient.
5. **Methods for Easy Manipulation**:
   - Java's `ArrayList` provides methods like `add()`, `remove()`, `get()`, and `set()` that make element management easier. It also supports iteration over the elements using enhanced for loops and iterators.
6. **No Preallocation of Memory**:
   - Since dynamic arrays grow as needed, they do not require the preallocation of memory for a fixed size, which is a limitation of regular arrays. This is useful in scenarios where the number of elements is unknown or unpredictable.

Disadvantages of Dynamic Arrays

1. **Resizing Overhead**:
    - When a dynamic array reaches its capacity, resizing involves allocating a new larger array and copying the elements from the old array to the new one, which takes O(n) time. Although this happens infrequently, it can be a costly operation if it occurs often in performance-sensitive applications.
2. **Insertions and Deletions (Middle/Beginning)**:
    - While appending elements to the end of a dynamic array is efficient, inserting or deleting elements at positions other than the end takes O(n) time, since it requires shifting elements to maintain the order of the array.
3. **Wasted Space**:
    - Dynamic arrays often allocate more space than needed to allow for future growth (e.g., doubling the capacity). This can lead to wasted memory if the array doesn't reach its full capacity.

Comparison with Static Arrays

Feature	Static Array	Dynamic Array (ArrayList)
Size	Fixed at initialization	Grows and shrinks dynamically
Memory Allocation	Predefined size	Adjusts size as needed
Access Time	O(1)	O(1)
Insert/Remove (End)	O(1)	O(1) (amortized)
Insert/Remove (Middle)	O(n)	O(n)
Wasted Space	None (fixed size)	Potentially some wasted memory
Resizing Cost	None	O(n) when resizing is needed

When to Use Dynamic Arrays (ArrayList)

- **When you need flexibility in size**: Use `ArrayList` when the size of your array may change during runtime.
- **Fast access and append**: If you need constant time access (via index) and frequent appending at the end, `ArrayList` is ideal.
- **When memory usage is not predictable**: Dynamic arrays save memory because they allocate space as needed, as opposed to static arrays which may waste space if initialized with excess capacity.

Conclusion

Dynamic arrays like Java's `ArrayList` offer the flexibility of automatic resizing, efficient element access, and support for adding or removing elements dynamically. They are highly useful in situations where the number of elements is unknown in advance, though they come with some overhead due to resizing and possible inefficiency for operations like inserting or deleting in the middle of the array.

Implement dynamic arrays and practice resizing.

=> Implementing a Dynamic Array in JavaScript

Dynamic arrays resize themselves as more elements are added. Below is an example of how you could implement a basic dynamic array in JavaScript. Since JavaScript arrays are already dynamic, this example will simulate a dynamic array using basic logic for resizing and managing capacity.

Dynamic Array Implementation

```
class DynamicArray {
    constructor() {
        this.size = 0; // Number of elements
        this.capacity = 2; // Initial capacity
        this.array = new Array(this.capacity); // Initial array with fixed capacity
    }

    // Method to add an element
    add(element) {
        // Check if we need to resize
        if (this.size === this.capacity) {
            this.resize();
        }
        // Add element to the array and increment size
        this.array[this.size] = element;
        this.size++;
    }

    // Method to resize the array (doubling the capacity)
    resize() {
        this.capacity *= 2; // Double the capacity
        let newArray = new Array(this.capacity); // Create a new array with the larger capacity

        // Copy elements from old array to the new array
        for (let i = 0; i < this.size; i++) {
            newArray[i] = this.array[i];
        }
        // Set the new array as the main array
        this.array = newArray;
    }

    // Method to get an element by index
    get(index) {
        if (index >= 0 && index < this.size) {
            return this.array[index];
        }
        return null;
    }

    // Method to remove an element (removes the last element)
    remove() {
        if (this.size > 0) {
            this.array[this.size - 1] = null; // Set last element to null
            this.size--; // Reduce size
        }
    }

    // Method to print the array
    print() {
        let result = [];
        for (let i = 0; i < this.size; i++) {
```

```
            result.push(this.array[i]);
        }
        console.log(result);
    }
}

// Testing the DynamicArray implementation
const dynamicArr = new DynamicArray();

dynamicArr.add(1);
dynamicArr.add(2);
dynamicArr.print(); // Output: [1, 2]

dynamicArr.add(3);  // Triggers resize
dynamicArr.print(); // Output: [1, 2, 3]

dynamicArr.add(4);
dynamicArr.add(5);  // Triggers another resize
dynamicArr.print(); // Output: [1, 2, 3, 4, 5]

console.log(dynamicArr.get(3)); // Output: 4 (element at index 3)

dynamicArr.remove();
dynamicArr.print(); // Output: [1, 2, 3, 4]
```

Key Points of the Implementation

1. **Initial Capacity**:
    - The array starts with an initial capacity of 2, and whenever it is filled, the array size is doubled using the `resize` function.
2. **Adding Elements**:
    - The `add` method adds elements to the array. When the array reaches capacity, it triggers the `resize` method to double the capacity before adding more elements.
3. **Resizing**:
    - The `resize` method doubles the size of the array, creating a new larger array and copying elements from the old array to the new one.
4. **Getting Elements**:
    - The `get` method allows you to retrieve elements by index.
5. **Removing Elements**:
    - The `remove` method removes the last element in the array.

Explanation of Resizing Process

- **Resizing Trigger**: The resizing process is triggered when the array reaches its capacity. This is done in the `add` method.
- **Doubling Capacity**: The `resize` method doubles the array's capacity, which provides some room for future elements.

- **Copying Elements**: After resizing, all existing elements are copied to the new array.

Practice Task

1. **Add elements beyond the initial capacity**:
    - Keep adding elements and observe the resizing.
    - Example: Try adding 10 elements in the `DynamicArray` and observe the capacity increase.
2. **Remove elements**:
    - Practice removing elements from the dynamic array.
3. **Test boundary cases**:
    - Try to access an element at an index that is out of bounds.
    - Try to remove elements when the array is empty.

By practicing the dynamic array concept, you can understand how arrays can be resized dynamically, which is a key feature in various programming languages like Java, Python, and JavaScript.

Day 6 : Subarrays and Subsequences

Explore subarray and subsequence concepts.

=> Subarrays and subsequences are fundamental concepts in array manipulation and are often used in various algorithms and data structure problems. Here's a detailed exploration of both concepts:

1. Subarray

A **subarray** is a contiguous segment of an array. It is defined by a starting and ending index, and it includes all the elements between these indices.

Characteristics of Subarrays:

- **Contiguous**: All elements in a subarray are consecutive in the original array.
- **Non-empty**: A subarray must contain at least one element.
- **Can be of any size**: The size of a subarray can range from 1 (a single element) to the length of the original array.

Example:

For the array `[1, 2, 3, 4]`:

- Possible subarrays include:
    - `[1]`
    - `[1, 2]`
    - `[1, 2, 3]`
    - `[1, 2, 3, 4]`
    - `[2]`

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

Common Problems Involving Subarrays:

- Finding the maximum sum of a subarray (Kadane's Algorithm).
- Finding the longest subarray with unique elements.
- Counting the number of subarrays that meet specific conditions (e.g., subarrays with a sum equal to a target value).

2. Subsequence

A **subsequence** is a sequence derived from another sequence by deleting some or no elements without changing the order of the remaining elements. Unlike subarrays, subsequences do not need to be contiguous.

Characteristics of Subsequences:

- **Non-contiguous**: Elements can be chosen from anywhere in the original array as long as their order is preserved.
- **Empty subsequence is valid**: The empty subsequence (no elements) is considered a valid subsequence.
- **Can be of any size**: The size of a subsequence can range from 0 (empty) to the length of the original array.

Example:

For the array `[1, 2, 3]`:

- Possible subsequences include:
  - `[]` (empty subsequence)
  - `[1]`
  - `[2]`
  - `[3]`
  - `[1, 2]`
  - `[1, 3]`
  - `[2, 3]`
  - `[1, 2, 3]`

Common Problems Involving Subsequences:

- Finding the longest increasing subsequence (LIS).
- Finding all possible subsequences of a given array.
- Determining if one sequence is a subsequence of another.

Comparing Subarrays and Subsequences

Feature	Subarray	Subsequence
Contiguity	Must be contiguous	Can be non-contiguous
Length	Must be at least 1	Can be 0 (empty) or up to the length of the array
Order	Elements must maintain their original order	Elements must maintain their original order
Example	`[2, 3]` in `[1, 2, 3, 4]`	`[1, 3]` in `[1, 2, 3, 4]`

Implementation Examples

Subarray Example: Finding Maximum Sum Subarray

Here's an example of finding the maximum sum of a contiguous subarray using Kadane's Algorithm:

```
function maxSubArray(nums) {
    let maxSoFar = nums[0];
    let maxEndingHere = nums[0];

    for (let i = 1; i < nums.length; i++) {
        maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}

const nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
console.log(maxSubArray(nums)); // Output: 6 (subarray [4, -1, 2, 1])
```

Subsequence Example: Finding Longest Increasing Subsequence

Here's how to find the longest increasing subsequence using dynamic programming:

```
function lengthOfLIS(nums) {
    if (nums.length === 0) return 0;

    const dp = new Array(nums.length).fill(1); // Initialize DP array

    for (let i = 1; i < nums.length; i++) {
        for (let j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
    }

    return Math.max(...dp); // Return the length of the longest increasing subsequence
}

const nums2 = [10, 9, 2, 5, 3, 7, 101, 18];
console.log(lengthOfLIS(nums2)); // Output: 4 (subsequence [2, 3, 7, 101])
```

Conclusion

Understanding the concepts of subarrays and subsequences is essential for solving many problems in algorithms and data structures. Knowing how to manipulate both structures effectively can help optimize solutions and improve the efficiency of your code.

Solve problems involving subarrays and subsequences.

=> Let's tackle a couple of problems involving subarrays and subsequences, along with their solutions.

Problem 1: Maximum Sum Subarray

Problem Statement: Given an integer array, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example: Input: `[-2,1,-3,4,-1,2,1,-5,4]` Output: `6` (the subarray `[4,-1,2,1]` has the largest sum)

Solution: We can use **Kadane's Algorithm** to solve this problem efficiently.

```
function maxSubArray(nums) {
    let maxSoFar = nums[0]; // To store the maximum sum found
    let maxEndingHere = nums[0]; // To store the maximum sum of subarray ending at current index

    for (let i = 1; i < nums.length; i++) {
        // Update maxEndingHere to include the current element or start a new subarray
        maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
        // Update maxSoFar if maxEndingHere is greater
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}

const nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
console.log(maxSubArray(nums)); // Output: 6
```

Problem 2: Longest Increasing Subsequence

Problem Statement: Given an integer array, return the length of the longest increasing subsequence.

Example: Input: `[10,9,2,5,3,7,101,18]` Output: `4` (the longest increasing subsequence is `[2,3,7,101]`)

Solution: We can use a dynamic programming approach to solve this problem.

```
function lengthOfLIS(nums) {
    if (nums.length === 0) return 0;

    const dp = new Array(nums.length).fill(1); // Initialize DP array

    for (let i = 1; i < nums.length; i++) {
        for (let j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
    }

    return Math.max(...dp); // Return the length of the longest increasing subsequence
}

const nums2 = [10, 9, 2, 5, 3, 7, 101, 18];
console.log(lengthOfLIS(nums2)); // Output: 4
```

Problem 3: Count Subarrays with Sum Equal to K

Problem Statement: Given an array of integers and an integer `k`, return the number of continuous subarrays whose sum equals `k`.

Example: Input: `nums = [1,1,1], k = 2` Output: `2` (the subarrays are `[1,1]` and `[1,1]`)

Solution: We can use a hashmap to store the cumulative sums and count the occurrences.

```
function subarraySum(nums, k) {
    let count = 0;
    let sum = 0;
    const sumCount = new Map();
    sumCount.set(0, 1); // To handle the case when the subarray starts from index 0

    for (let num of nums) {
        sum += num; // Update the cumulative sum
        // Check if there's a subarray (ending at the current index) with sum equal to k
        if (sumCount.has(sum - k)) {
            count += sumCount.get(sum - k); // Increment the count
        }
        // Update the hashmap with the current cumulative sum
        sumCount.set(sum, (sumCount.get(sum) || 0) + 1);
    }

    return count;
}

const nums3 = [1, 1, 1];
const k = 2;
console.log(subarraySum(nums3, k)); // Output: 2
```

Problem 4: Find All Subsequences of an Array

Problem Statement: Given an array, find all possible subsequences (including the empty subsequence).

Example: Input: `[1, 2]` Output: `[], [1], [2], [1, 2]`

Solution: We can use a recursive approach or backtracking to generate all subsequences.

```
function generateSubsequences(nums) {
    const result = [];

    const backtrack = (start, current) => {
        result.push([...current]); // Add the current subsequence to the result
        for (let i = start; i < nums.length; i++) {
            current.push(nums[i]); // Include nums[i] in the current subsequence
            backtrack(i + 1, current); // Move to the next element
            current.pop(); // Backtrack and remove nums[i]
        }
    };

    backtrack(0, []); // Start the recursion
    return result;
}

const nums4 = [1, 2];
console.log(generateSubsequences(nums4));
// Output: [ [], [ 1 ], [ 2 ], [ 1, 2 ] ]
```

Summary

- **Maximum Sum Subarray**: Using Kadane's Algorithm.
- **Longest Increasing Subsequence**: Dynamic programming.
- **Count Subarrays with Sum Equal to K**: Using a hashmap for cumulative sums.
- **Find All Subsequences**: Using backtracking.

These problems showcase how subarrays and subsequences can be manipulated and the various techniques that can be employed to solve them effectively. If you need more problems or explanations on a specific topic, feel free to ask!

Day 7 : Array Rotation and Reversal

Learn techniques for rotating and reversing arrays.

=> Rotating and reversing arrays are common operations that can be performed using different techniques. Below, I'll explain both concepts and provide examples in JavaScript.

1. Rotating an Array

Rotating an array means shifting the elements of the array either to the left or right by a given number of positions.

a. Rotate Right

Example: Given an array and a number `k`, rotate the array to the right by `k` positions.

Input: `[1, 2, 3, 4, 5], k = 2` **Output**: `[4, 5, 1, 2, 3]`

Solution:

```
function rotateRight(arr, k) {
    const n = arr.length;
    k = k % n; // Handle cases where k is greater than n
    if (k === 0) return arr; // No rotation needed

    // Step 1: Reverse the whole array
    reverse(arr, 0, n - 1);
    // Step 2: Reverse the first k elements
    reverse(arr, 0, k - 1);
    // Step 3: Reverse the rest of the elements
    reverse(arr, k, n - 1);

    return arr;
}

function reverse(arr, start, end) {
    while (start < end) {
        [arr[start], arr[end]] = [arr[end], arr[start]]; // Swap elements
        start++;
        end--;
    }
}

const arr1 = [1, 2, 3, 4, 5];
const k1 = 2;
console.log(rotateRight(arr1, k1)); // Output: [4, 5, 1, 2, 3]
```

b. Rotate Left

Example: Given an array and a number `k`, rotate the array to the left by `k` positions.

Input: `[1, 2, 3, 4, 5], k = 2` **Output**: `[3, 4, 5, 1, 2]`

Solution:

```
function rotateLeft(arr, k) {
    const n = arr.length;
    k = k % n; // Handle cases where k is greater than n
    if (k === 0) return arr; // No rotation needed

    // Step 1: Reverse the first n-k elements
    reverse(arr, 0, n - k - 1);
    // Step 2: Reverse the last k elements
    reverse(arr, n - k, n - 1);
    // Step 3: Reverse the whole array
    reverse(arr, 0, n - 1);

    return arr;
}

const arr2 = [1, 2, 3, 4, 5];
const k2 = 2;
console.log(rotateLeft(arr2, k2)); // Output: [3, 4, 5, 1, 2]
```

2. Reversing an Array

Reversing an array means changing the order of elements such that the first element becomes the last and the last becomes the first.

Example: Reverse the array `[1, 2, 3, 4, 5]`.

Input: `[1, 2, 3, 4, 5]` **Output**: `[5, 4, 3, 2, 1]`

Solution:

```
function reverseArray(arr) {
    let start = 0;
    let end = arr.length - 1;

    while (start < end) {
        [arr[start], arr[end]] = [arr[end], arr[start]]; // Swap elements
        start++;
        end--;
    }

    return arr;
}

const arr3 = [1, 2, 3, 4, 5];
console.log(reverseArray(arr3)); // Output: [5, 4, 3, 2, 1]
```

Summary

- **Rotating an Array**:
    - **Right Rotation**: Use three reversals to achieve the rotation.
    - **Left Rotation**: Also use three reversals, adjusting indices accordingly.

- **Reversing an Array**: A simple two-pointer technique where you swap elements from the beginning and end until they meet in the middle.

These techniques are efficient and utilize constant space, making them suitable for large arrays. If you have further questions or want to explore more complex scenarios, let me know!

Practice problems involving array rotation and reversal.

=> Here are some practice problems involving array rotation and reversal. Each problem includes a description, sample input/output, and solutions using JavaScript.

Problem 1: Rotate Array to the Right

Description: Given an array of integers and a non-negative integer `k`, rotate the array to the right by `k` steps.

Input:

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

Output:

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

Solution:

```
function rotateRight(arr, k) {
    const n = arr.length;
    k = k % n; // Handle cases where k is greater than n
    if (k === 0) return arr;

    reverse(arr, 0, n - 1);
    reverse(arr, 0, k - 1);
    reverse(arr, k, n - 1);
    return arr;
}

function reverse(arr, start, end) {
    while (start < end) {
        [arr[start], arr[end]] = [arr[end], arr[start]];
        start++;
        end--;
    }
}

const arr1 = [1, 2, 3, 4, 5];
console.log(rotateRight(arr1, 2)); // Output: [4, 5, 1, 2, 3]
```

Problem 2: Rotate Array to the Left

Description: Given an array of integers and a non-negative integer `k`, rotate the array to the left by `k` steps.

Input:

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

Output:

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

Solution:

```
function rotateLeft(arr, k) {
    const n = arr.length;
    k = k % n; // Handle cases where k is greater than n
    if (k === 0) return arr;

    reverse(arr, 0, n - k - 1);
    reverse(arr, n - k, n - 1);
    reverse(arr, 0, n - 1);
    return arr;
}

const arr2 = [1, 2, 3, 4, 5];
console.log(rotateLeft(arr2, 2)); // Output: [3, 4, 5, 1, 2]
```

Problem 3: Reverse an Array

Description: Write a function that reverses an array in place.

Input:

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

Output:

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

Solution:

```
function reverseArray(arr) {
    let start = 0;
    let end = arr.length - 1;

    while (start < end) {
        [arr[start], arr[end]] = [arr[end], arr[start]];
        start++;
        end--;
    }

    return arr;
}

const arr3 = [1, 2, 3, 4, 5];
console.log(reverseArray(arr3)); // Output: [5, 4, 3, 2, 1]
```

Problem 4: Rotate Array Using Extra Space

Description: Implement an array rotation function that uses additional space instead of reversing the array.

Input:

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

Output:

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

Solution:

```
function rotateWithExtraSpace(arr, k) {
    const n = arr.length;
    k = k % n; // Handle cases where k is greater than n
    const rotated = new Array(n);

    for (let i = 0; i < n; i++) {
        rotated[(i + k) % n] = arr[i];
    }

    return rotated;
}

const arr4 = [1, 2, 3, 4, 5];
console.log(rotateWithExtraSpace(arr4, 2)); // Output: [4, 5, 1, 2, 3]
```

Problem 5: Reverse Words in a Sentence

Description: Given a sentence as an array of characters, reverse the characters of each word while still preserving whitespace and initial word order.

Input:

```
["H", "e", "l", "l", "o", " ", "W", "o", "r", "l", "d"]
```

Output:

```
["o", "l", "l", "e", "H", " ", "d", "l", "r", "o", "W"]
```

Solution:

```
function reverseWords(arr) {
    let start = 0;

    for (let end = 0; end <= arr.length; end++) {
        if (end === arr.length || arr[end] === ' ') {
            reverse(arr, start, end - 1);
            start = end + 1;
        }
    }

    reverse(arr, 0, arr.length - 1); // Reverse the whole array
    return arr;
}

const sentence = ["H", "e", "l", "l", "o", " ", "W", "o", "r", "l", "d"];
console.log(reverseWords(sentence)); // Output: ["d", "l", "r", "o", "W", " ", "o", "l", "l", "e", "H"]
```

Summary of Practice Problems

These practice problems cover array rotations and reversals, using both in-place and extra space techniques. You can enhance your understanding and coding skills by implementing these problems and experimenting with different inputs! If you have any specific topics or problems in mind, feel free to ask!

Day 8 : Duplicate and Missing Elements

Study approaches to find duplicate and missing elements in an array.

=> Finding duplicate and missing elements in an array is a common problem in programming, particularly in the context of algorithm and data structure studies. Here's a breakdown of various approaches to tackle these problems, along with sample code snippets in JavaScript.

1. Finding Duplicates in an Array

a. Using a Set

Approach: Utilize a Set to track seen elements. If an element is already in the Set, it's a duplicate.

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

Example:

```
function findDuplicates(arr) {
    const seen = new Set();
    const duplicates = [];

    for (const num of arr) {
        if (seen.has(num)) {
            duplicates.push(num);
        } else {
            seen.add(num);
        }
    }

    return duplicates;
}

const arr1 = [1, 2, 3, 4, 4, 5, 1, 2];
console.log(findDuplicates(arr1)); // Output: [4, 1, 2]
```

b. Using Frequency Counter Object

Approach: Use an object to count occurrences of each element. Elements that appear more than once are duplicates.

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

Example:

```
function findDuplicates(arr) {
    const count = {};
    const duplicates = [];

    for (const num of arr) {
        count[num] = (count[num] || 0) + 1;
    }

    for (const [key, value] of Object.entries(count)) {
        if (value > 1) {
            duplicates.push(Number(key));
        }
    }

    return duplicates;
}

const arr2 = [1, 2, 3, 4, 4, 5, 1, 2];
console.log(findDuplicates(arr2)); // Output: [4, 1, 2]
```

2. Finding Missing Elements in an Array

a. Using the Expected Sum Formula

Approach: Calculate the expected sum of the first `n` natural numbers and compare it with the actual sum of the array. The difference is the missing number.

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

Example:

```
function findMissing(arr, n) {
    const expectedSum = (n * (n + 1)) / 2;
    const actualSum = arr.reduce((acc, num) => acc + num, 0);
    return expectedSum - actualSum;
}

const arr3 = [1, 2, 4, 5, 6]; // Missing 3
const n = 6; // Length of the complete array
console.log(findMissing(arr3, n)); // Output: 3
```

b. Using a Set

Approach: Store the elements in a Set and check which number from 1 to `n` is missing.

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

Example:

```
function findMissing(arr, n) {
    const seen = new Set(arr);
    for (let i = 1; i <= n; i++) {
        if (!seen.has(i)) {
            return i;
        }
    }
    return -1; // If no missing number found
}

const arr4 = [1, 2, 3, 5]; // Missing 4
const n2 = 5; // Length of the complete array
console.log(findMissing(arr4, n2)); // Output: 4
```

3. Finding Both Duplicates and Missing Elements

Approach: Use a frequency counter object to count occurrences. Iterate through the expected range and check against the counts to identify duplicates and missing numbers.

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

Example:

```
function findDuplicatesAndMissing(arr, n) {
    const count = {};
    const duplicates = [];
    let missing = -1;

    for (const num of arr) {
        count[num] = (count[num] || 0) + 1;
    }

    for (let i = 1; i <= n; i++) {
        if (count[i] > 1) {
            duplicates.push(i);
        } else if (!count[i]) {
            missing = i;
        }
    }

    return { duplicates, missing };
}

const arr5 = [1, 2, 2, 4, 5]; // Duplicate: 2, Missing: 3
const n3 = 5; // Length of the complete array
console.log(findDuplicatesAndMissing(arr5, n3)); // Output: { duplicates: [2], missing: 3 }
```

Summary

- **Finding Duplicates**: Use a Set or frequency counter.

- **Finding Missing Elements**: Use the expected sum formula or a Set.
- **Finding Both**: Combine the above methods for a single solution.

These approaches can help in efficiently solving problems related to duplicates and missing elements in arrays. If you have further questions or need more examples, feel free to ask!

Solve problems related to duplicate and missing values.

=> Sure! Let's tackle a couple of problems related to duplicates and missing values in arrays. I'll provide examples and solutions for each problem.

Problem 1: Find All Duplicates in an Array

Description: Given an array of integers, where each integer is in the range from 1 to n (inclusive) and n is the size of the array, some elements appear twice, and others appear once. Return a list of all the integers that appear twice.

Example:

- **Input**: `[4, 3, 2, 7, 8, 2, 3, 1]`
- **Output**: `[2, 3]`

Solution:

```
function findAllDuplicates(arr) {
    const duplicates = [];
    const seen = new Set();

    for (const num of arr) {
        if (seen.has(num)) {
            duplicates.push(num);
        } else {
            seen.add(num);
        }
    }

    return duplicates;
}

const input1 = [4, 3, 2, 7, 8, 2, 3, 1];
console.log(findAllDuplicates(input1)); // Output: [2, 3]
```

Problem 2: Find the Missing Number

Description: Given an array containing `n` distinct numbers taken from 0, 1, 2, ..., n, find the one number that is missing from the array.

Example:

- **Input**: `[3, 0, 1]`
- **Output**: `2`

Solution:

```
function findMissingNumber(arr) {
    const n = arr.length;
    const expectedSum = (n * (n + 1)) / 2; // Sum of first n natural numbers
    const actualSum = arr.reduce((acc, num) => acc + num, 0); // Sum of elements in the array

    return expectedSum - actualSum; // The missing number
}

const input2 = [3, 0, 1];
console.log(findMissingNumber(input2)); // Output: 2
```

Problem 3: Find Duplicates and the Missing Number

Description: Given an array containing `n` integers where each integer is in the range from 1 to n. Some numbers appear twice and some appear once. Find the missing number and the duplicate number.

Example:

- **Input**: `[1, 2, 2, 4]`
- **Output**: `{ duplicate: 2, missing: 3 }`

Solution:

```
function findDuplicateAndMissing(arr) {
    const count = {};
    let duplicate, missing;

    // Count occurrences of each number
    for (const num of arr) {
        count[num] = (count[num] || 0) + 1;
    }

    // Find duplicate and missing numbers
    for (let i = 1; i <= arr.length; i++) {
        if (count[i] === 2) {
            duplicate = i;
        } else if (!count[i]) {
            missing = i;
        }
    }

    return { duplicate, missing };
}

const input3 = [1, 2, 2, 4];
console.log(findDuplicateAndMissing(input3)); // Output: { duplicate: 2, missing: 3 }
```

Summary

- **Finding Duplicates**: Use a Set or an object to count occurrences.
- **Finding Missing Numbers**: Use the expected sum formula or counting methods.
- **Combined Approach**: Count occurrences and check for duplicates and missing numbers simultaneously.

Feel free to ask if you'd like to see more problems or solutions!

Day 9 : Frequency Count and Prefix Sum

Explore techniques like frequency counting and prefix sum.

=> Sure! Let’s explore the techniques of **frequency counting** and **prefix sum** in the context of solving problems related to arrays.

1. Frequency Counting

Description: Frequency counting is a technique used to count the number of occurrences of each distinct element in an array. This technique is useful for problems involving duplicates, finding the most frequent element, or checking the existence of elements.

Example Problems

Problem 1: Find the Most Frequent Element

Input: `[1, 3, 2, 2, 3, 1, 1, 1]` **Output**: `1` (most frequent)

Solution:

```
function mostFrequentElement(arr) {
    const frequency = {};

    for (const num of arr) {
        frequency[num] = (frequency[num] || 0) + 1;
    }

    let maxCount = 0;
    let mostFrequent;

    for (const [num, count] of Object.entries(frequency)) {
        if (count > maxCount) {
            maxCount = count;
            mostFrequent = num;
        }
    }

    return mostFrequent;
}

const input1 = [1, 3, 2, 2, 3, 1, 1, 1];
console.log(mostFrequentElement(input1)); // Output: 1
```

Problem 2: Check if Two Arrays Have the Same Elements

Input: `[1, 2, 3], [3, 2, 1]` **Output**: `true`

Solution:

```
function haveSameElements(arr1, arr2) {
    if (arr1.length !== arr2.length) return false;

    const frequency = {};

    for (const num of arr1) {
        frequency[num] = (frequency[num] || 0) + 1;
    }

    for (const num of arr2) {
        if (!frequency[num]) {
            return false;
        }
        frequency[num]--;
    }

    return true;
}

const input2a = [1, 2, 3];
const input2b = [3, 2, 1];
console.log(haveSameElements(input2a, input2b)); // Output: true
```

2. Prefix Sum

Description: The prefix sum technique is used to compute the cumulative sum of an array. It allows for efficient range sum queries and can be applied in various problems, such as finding subarray sums.

Example Problems

Problem 1: Range Sum Query

Given an array, efficiently calculate the sum of elements between two indices.

Input: `arr = [1, 2, 3, 4, 5]`, query `(1, 3)` (sum from index 1 to 3) **Output**: `9` (2 + 3 + 4)

Solution:

```
function prefixSum(arr) {
    const prefix = [0]; // Prefix sum array with an initial zero

    for (let i = 0; i < arr.length; i++) {
        prefix[i + 1] = prefix[i] + arr[i];
    }

    return prefix;
}

function rangeSum(arr, start, end) {
    const prefix = prefixSum(arr);
    return prefix[end + 1] - prefix[start]; // Adjusting for zero-based index
}

const input3 = [1, 2, 3, 4, 5];
console.log(rangeSum(input3, 1, 3)); // Output: 9 (2 + 3 + 4)
```

Problem 2: Number of Subarrays with a Given Sum

Input: `arr = [1, 2, 3, 3, 2, 1]`, target sum `6` **Output**: `3` (subarrays: [1, 2, 3], [3, 3], [2, 1, 3])

Solution:

```
function countSubarraysWithSum(arr, targetSum) {
    const prefix = prefixSum(arr);
    const count = {};
    let totalCount = 0;

    for (let i = 0; i < prefix.length; i++) {
        if (prefix[i] === targetSum) {
            totalCount++;
        }
        if (count[prefix[i] - targetSum]) {
            totalCount += count[prefix[i] - targetSum];
        }
        count[prefix[i]] = (count[prefix[i]] || 0) + 1;
    }

    return totalCount;
}

const input4 = [1, 2, 3, 3, 2, 1];
const targetSum = 6;
console.log(countSubarraysWithSum(input4, targetSum)); // Output: 3
```

Summary

- **Frequency Counting**:
    - Useful for counting occurrences, finding most frequent elements, and checking element existence.

- Typically implemented using an object or a `Map` to store counts.

- **Prefix Sum**:
  - Useful for efficiently calculating range sums and answering queries related to subarrays.
  - Typically involves creating a new array that holds cumulative sums.

Both techniques are powerful tools for solving array-related problems efficiently. If you have specific problems in mind or need further examples, feel free to ask!

Practice problems that involve counting elements and finding prefix sums.

=> Here are the solutions to the problems provided, along with explanations for each approach.

Problem 1: Count Unique Elements

```
function countUniqueElements(arr) {
    const uniqueElements = new Set(arr);
    return uniqueElements.size;
}

// Example usage
const input1 = [1, 2, 2, 3, 4, 4, 4, 5];
console.log(countUniqueElements(input1)); // Output: 5
```

Problem 2: Find the First Repeated Element

```
function firstRepeatedElement(arr) {
    const seen = new Set();
    for (const num of arr) {
        if (seen.has(num)) {
            return num;
        }
        seen.add(num);
    }
    return null; // Return null if no repeated element is found
}

// Example usage
const input2 = [1, 2, 3, 4, 2, 5];
console.log(firstRepeatedElement(input2)); // Output: 2
```

Problem 3: Prefix Sum of an Array

```
function prefixSum(arr) {
    const result = [];
    let currentSum = 0;
    for (const num of arr) {
        currentSum += num;
        result.push(currentSum);
    }
    return result;
}

// Example usage
const input3 = [1, 2, 3, 4];
console.log(prefixSum(input3)); // Output: [1, 3, 6, 10]
```

Problem 4: Subarray Sum Equals K

```
function subarraySum(arr, k) {
    const prefixSumCounts = new Map();
    prefixSumCounts.set(0, 1); // Initialize with sum 0
    let count = 0;
    let currentSum = 0;

    for (const num of arr) {
        currentSum += num;
        if (prefixSumCounts.has(currentSum - k)) {
            count += prefixSumCounts.get(currentSum - k);
        }
        prefixSumCounts.set(currentSum, (prefixSumCounts.get(currentSum) || 0) + 1);
    }

    return count;
}

// Example usage
const input4 = [1, 1, 1];
const k4 = 2;
console.log(subarraySum(input4, k4)); // Output: 2
```

Problem 5: Count Occurrences of an Element

```
function countOccurrences(arr, element) {
    let count = 0;
    for (const num of arr) {
        if (num === element) {
            count++;
        }
    }
    return count;
}

// Example usage
const input5 = [1, 2, 3, 1, 2, 1];
const element5 = 1;
console.log(countOccurrences(input5, element5)); // Output: 3
```

Problem 6: Maximum Prefix Sum

```
function maximumPrefixSum(arr) {
    let maxSum = -Infinity;
    let currentSum = 0;

    for (const num of arr) {
        currentSum += num;
        maxSum = Math.max(maxSum, currentSum);
    }

    return maxSum;
}

// Example usage
const input6 = [-1, 2, 3, -2, 5];
console.log(maximumPrefixSum(input6)); // Output: 8
```

Problem 7: Find the Number of Distinct Elements in Subarrays

```
function countDistinctInSubarrays(arr, k) {
    const result = [];
    const frequency = new Map();

    for (let i = 0; i < arr.length; i++) {
        if (i >= k) {
            // Remove the element that is sliding out of the window
            const slidingOut = arr[i - k];
            frequency.set(slidingOut, frequency.get(slidingOut) - 1);
            if (frequency.get(slidingOut) === 0) {
                frequency.delete(slidingOut);
            }
        }

        // Add the new element into the window
        const slidingIn = arr[i];
        frequency.set(slidingIn, (frequency.get(slidingIn) || 0) + 1);

        if (i >= k - 1) {
            result.push(frequency.size); // Add count of distinct elements
        }
    }

    return result;
}

// Example usage
const input7 = [1, 2, 1, 3, 4];
const k7 = 3;
console.log(countDistinctInSubarrays(input7, k7)); // Output: [2, 3, 3]
```

Problem 8: Count of Subarrays with Sum Less Than K

```
function countSubarraysWithSumLessThanK(arr, k) {
    let count = 0;
    let currentSum = 0;
    let start = 0;

    for (let end = 0; end < arr.length; end++) {
        currentSum += arr[end];

        // Shrink the window until the currentSum is less than k
        while (currentSum >= k && start <= end) {
            currentSum -= arr[start];
            start++;
        }

        // All subarrays ending at 'end' and starting from 'start' to 'end' are valid
        count += (end - start + 1);
    }

    return count;
}

// Example usage
const input8 = [1, 2, 3, 4];
const k8 = 7;
console.log(countSubarraysWithSumLessThanK(input8, k8)); // Output: 6
```

Feel free to test these solutions with other input values or modify them as needed! If you have any questions about how they work or if you need further assistance, just let me know!

Day 10 : Review and Advanced Problems

1. Set Matrix Zeroes

=> To solve the "Set Matrix Zeroes" problem, we can use an approach that first identifies which rows and columns need to be set to zero and then modifies the matrix accordingly. Here's a step-by-step breakdown of the algorithm, followed by the implementation:

Steps:

1. **Identify Rows and Columns to be Zeroed**: Use two sets to keep track of which rows and columns need to be zeroed. We can traverse the matrix to find all the zero elements and record their respective row and column indices.
2. **Set the Rows to Zero**: Iterate through the identified rows and set all elements in those rows to zero.
3. **Set the Columns to Zero**: Iterate through the identified columns and set all elements in those columns to zero.

Implementation:

Here's how you can implement this in JavaScript:

```
function setMatrixZeroes(matrix) {
    const rows = matrix.length;
    const cols = matrix[0].length;

    const zeroRows = new Set();
    const zeroCols = new Set();

    // Step 1: Identify rows and columns that need to be zeroed
    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            if (matrix[i][j] === 0) {
                zeroRows.add(i); // Add row index to zeroRows
                zeroCols.add(j); // Add column index to zeroCols
            }
        }
    }

    // Step 2: Set the identified rows to zero
    for (let i = 0; i < rows; i++) {
        if (zeroRows.has(i)) {
            for (let j = 0; j < cols; j++) {
                matrix[i][j] = 0;
            }
        }
    }

    // Step 3: Set the identified columns to zero
    for (let j = 0; j < cols; j++) {
        if (zeroCols.has(j)) {
            for (let i = 0; i < rows; i++) {
                matrix[i][j] = 0;
            }
        }
    }
}

// Example usage:
const matrix1 = [[1, 1, 1], [1, 0, 1], [1, 1, 1]];
setMatrixZeroes(matrix1);
console.log(matrix1); // Output: [[1, 0, 1], [0, 0, 0], [1, 0, 1]]

const matrix2 = [[0, 1, 2, 0], [3, 4, 5, 2], [1, 3, 1, 5]];
setMatrixZeroes(matrix2);
console.log(matrix2); // Output: [[0, 0, 0, 0], [0, 4, 5, 0], [0, 3, 1, 0]]
```

Explanation:

- **Step 1**: We iterate through the matrix and populate the `zeroRows` and `zeroCols` sets with the indices of the rows and columns that contain at least one zero.
- **Step 2**: We iterate through the matrix again, checking if each row index is in `zeroRows`. If it is, we set all elements in that row to zero.

- **Step 3**: We repeat the process for columns using the `zeroCols` set.

Complexity Analysis:

- **Time Complexity**: O(m * n), where m is the number of rows and n is the number of columns. We traverse the matrix twice.
- **Space Complexity**: O(m + n) for the two sets used to store the row and column indices.

Feel free to test the function with other matrices or adjust it according to your needs! If you have any further questions or need additional help, let me know!

2. Two Sum

=> To solve the problem of finding two indices in an array that add up to a specified target, we can use a hash map (object in JavaScript) to store the indices of the numbers as we iterate through the array. This allows us to find the complement of each number efficiently. Here's how the algorithm works:

Steps:

1. **Initialize a Map**: Create an empty map to store the value and its corresponding index as we iterate through the array.
2. **Iterate Through the Array**: For each element in the array:
    - Calculate its complement by subtracting the current element from the target.
    - Check if the complement exists in the map. If it does, return the indices of the current element and the complement.
    - If the complement does not exist, add the current element and its index to the map.

Implementation:

Here's how you can implement this in JavaScript:

```
function twoSum(nums, target) {
    const numMap = new Map(); // Create a map to store numbers and their indices

    for (let i = 0; i < nums.length; i++) {
        const complement = target - nums[i]; // Calculate the complement

        // Check if the complement is already in the map
        if (numMap.has(complement)) {
            return [numMap.get(complement), i]; // Return the indices
        }

        // Add the current number and its index to the map
        numMap.set(nums[i], i);
    }

    // Return an empty array if no solution is found (though the problem guarantees one solution)
    return [];
}

// Example usage:
console.log(twoSum([2, 7, 11, 15], 9)); // Output: [0, 1]
console.log(twoSum([3, 2, 4], 6));      // Output: [1, 2]
console.log(twoSum([3, 3], 6));         // Output: [0, 1]
```

Explanation:

- **Map Initialization**: We use a `Map` to store the numbers we have seen so far along with their indices.
- **Finding the Complement**: For each number in the array, we check if the complement (the number needed to reach the target) exists in our map.
- **Returning the Result**: If we find a match, we return the indices of the complement and the current number.

Complexity Analysis:

- **Time Complexity**: O(n), where n is the number of elements in the array. We traverse the array only once.
- **Space Complexity**: O(n), as we may store up to n elements in the map.

This approach is efficient and straightforward, allowing us to find the solution in linear time. Let me know if you have any questions or need further assistance!

3. Best Time to Buy and Sell Stock II

=> To solve the problem of maximizing profit from stock prices given in an array, we can follow a simple approach where we buy and sell the stock on every profitable day. The strategy involves iterating through the prices and calculating the profit whenever there is a price increase from one day to the next.

Approach:

1. **Initialize Profit**: Start with a profit variable set to zero.
2. **Iterate Through Prices**: For each day in the prices array, check if the price of the current day is higher than the previous day. If it is, add the difference to the profit.
3. **Return Total Profit**: At the end of the iteration, return the total profit.

Implementation:

Here's the JavaScript code that implements this approach:

```
function maxProfit(prices) {
    let profit = 0; // Initialize profit to 0

    // Iterate through the prices
    for (let i = 1; i < prices.length; i++) {
        // If the price has increased from the previous day, add the profit
        if (prices[i] > prices[i - 1]) {
            profit += prices[i] - prices[i - 1];
        }
    }

    return profit; // Return the total profit
}

// Example usage:
console.log(maxProfit([7, 1, 5, 3, 6, 4])); // Output: 7
console.log(maxProfit([1, 2, 3, 4, 5]));    // Output: 4
console.log(maxProfit([7, 6, 4, 3, 1]));    // Output: 0
```

Explanation:

- **Profit Calculation**: We only add to our profit when we find a price that is higher than the price on the previous day. This simulates buying on the previous day and selling on the current day.
- **No Holding Restriction**: Since you can buy and sell on the same day, we treat every price increase as a potential profit opportunity.

Complexity Analysis:

- **Time Complexity**: O(n), where n is the number of days (length of the prices array). We only need a single pass through the prices.
- **Space Complexity**: O(1), since we are using a constant amount of extra space for the profit variable.

This method efficiently calculates the maximum profit you can achieve with the given stock prices. If you have further questions or need additional help, feel free to ask!

4. Best Time to Buy and Sell Stock

=> To solve the problem of maximizing profit from stock prices with the condition that you can buy on one day and sell on a later day, we can use a simple approach that tracks the minimum price seen so far and calculates the maximum profit accordingly.

Approach:

1. **Initialize Variables**: Start with a variable to keep track of the minimum price and another for the maximum profit.
2. **Iterate Through Prices**: For each price in the array:
    - Update the minimum price if the current price is lower than the minimum price.
    - Calculate the potential profit by subtracting the minimum price from the current price.
    - Update the maximum profit if the calculated profit is greater than the current maximum profit.
3. **Return Maximum Profit**: At the end of the iteration, return the maximum profit.

Implementation:

Here's how this can be implemented in JavaScript:

```
function maxProfit(prices) {
    let minPrice = Infinity; // Start with a high minimum price
    let maxProfit = 0; // Initialize max profit to 0

    for (let price of prices) {
        // Update the minimum price if the current price is lower
        if (price < minPrice) {
            minPrice = price;
        }

        // Calculate potential profit
        const profit = price - minPrice;

        // Update the maximum profit if the current profit is higher
        if (profit > maxProfit) {
            maxProfit = profit;
        }
    }

    return maxProfit; // Return the maximum profit
}

// Example usage:
console.log(maxProfit([7, 1, 5, 3, 6, 4])); // Output: 5
console.log(maxProfit([7, 6, 4, 3, 1]));    // Output: 0
```

Explanation:

- **Track Minimum Price**: As we iterate through the prices, we continuously check if the current price is the lowest we've seen, updating our `minPrice` accordingly.
- **Calculate Profit**: For each price, we calculate the profit that could be made if we bought at the `minPrice` and sold at the current price.
- **Update Maximum Profit**: If this potential profit exceeds our recorded `maxProfit`, we update it.

Complexity Analysis:

- **Time Complexity**: O(n), where n is the number of days (length of the prices array). We only need a single pass through the prices.
- **Space Complexity**: O(1), since we are using a constant amount of extra space for the `minPrice` and `maxProfit` variables.

This approach efficiently calculates the maximum profit achievable from buying and selling stock under the given constraints. If you have any more questions or need further clarification, feel free to ask!

5. Sort Colors

=> To solve the problem of sorting an array of integers representing colors (0 for red, 1 for white, and 2 for blue), we can utilize the **Dutch National Flag algorithm**. This algorithm allows us to sort the array in one pass (O(n) time complexity) and uses constant space (O(1) space complexity).

Approach:

1. **Initialize Pointers**:
    - `low` pointer to track the next position for `0` (red).
    - `mid` pointer to track the current element.
    - `high` pointer to track the next position for `2` (blue).
2. **Iterate Through the Array**:
    - If the element at the `mid` pointer is `0`, swap it with the element at the `low` pointer and increment both `low` and `mid`.
    - If the element at the `mid` pointer is `1`, just increment `mid`.
    - If the element at the `mid` pointer is `2`, swap it with the element at the `high` pointer and decrement `high` (do not increment `mid` immediately because the swapped value needs to be checked).
3. **Continue Until the `mid` Pointer Exceeds the `high` Pointer**.

Implementation:

Here's how this can be implemented in JavaScript:

```
function sortColors(nums) {
    let low = 0;        // Pointer for the next position of 0
    let mid = 0;        // Current element pointer
    let high = nums.length - 1; // Pointer for the next position of 2

    while (mid <= high) {
        if (nums[mid] === 0) {
            // Swap nums[low] and nums[mid]
            [nums[low], nums[mid]] = [nums[mid], nums[low]];
            low++;
            mid++;
        } else if (nums[mid] === 1) {
            mid++;
        } else {
            // Swap nums[mid] and nums[high]
            [nums[mid], nums[high]] = [nums[high], nums[mid]];
            high--;
        }
    }
}

// Example usage:
const nums1 = [2, 0, 2, 1, 1, 0];
sortColors(nums1);
console.log(nums1); // Output: [0, 0, 1, 1, 2, 2]

const nums2 = [2, 0, 1];
sortColors(nums2);
console.log(nums2); // Output: [0, 1, 2]
```

Explanation:

- **Pointer Initialization**: The `low`, `mid`, and `high` pointers are initialized to point to the start, start, and end of the array, respectively.
- **Element Check and Swap**:
  - If the current element (`nums[mid]`) is `0`, it is swapped with the element at the `low` pointer, and both pointers are moved forward.
  - If the current element is `1`, we just move the `mid` pointer forward.
  - If the current element is `2`, we swap it with the element at the `high` pointer, moving `high` backwards.

Complexity Analysis:

- **Time Complexity**: O(n), where n is the number of elements in the array. We make a single pass through the array.
- **Space Complexity**: O(1), as we are sorting in place and using a constant amount of additional space for pointers.

This algorithm efficiently sorts the colors in a single pass. If you have any further questions or need additional explanations, feel free to ask!

6. Find All Duplicates in an Array

=> To solve the problem of finding all integers that appear twice in the given integer array, we can use the input array itself to track which numbers we've seen. Since the integers are in the range ([1, n]) (where (n) is the length of the array), we can utilize the indices of the array to mark the numbers we've encountered.

Approach:

1. **Iterate Through the Array**: For each number in the array, calculate its corresponding index by taking the absolute value of the number and subtracting 1 (because array indices start at 0).
2. **Check the Value at the Index**:
    - If the value at that index is negative, it means we have already seen this number, so we add it to the result list.
    - If it is positive, we negate the value at that index to mark that we have seen this number.
3. **Return the Result**: At the end of the loop, return the list of duplicates.

Implementation:

Here's how you can implement this in JavaScript:

```
function findDuplicates(nums) {
    const duplicates = [];

    for (let i = 0; i < nums.length; i++) {
        // Calculate the index for the current number
        const index = Math.abs(nums[i]) - 1;

        // Check if the number at that index is already negative
        if (nums[index] < 0) {
            duplicates.push(index + 1); // Add the number to the result (convert back to 1-based)
        } else {
            nums[index] = -nums[index]; // Mark the number as seen
        }
    }

    return duplicates;
}

// Example usage:
const nums1 = [4, 3, 2, 7, 8, 2, 3, 1];
console.log(findDuplicates(nums1)); // Output: [2, 3]

const nums2 = [1, 1, 2];
console.log(findDuplicates(nums2)); // Output: [1]

const nums3 = [1];
console.log(findDuplicates(nums3)); // Output: []
```

Explanation:

- **Index Calculation**: We compute the index for each number as `Math.abs(nums[i]) - 1`. We use `Math.abs()` to handle cases where we have already negated the value at that index.
- **Negation**: By negating the value at the calculated index, we mark that we have seen the number corresponding to that index.

- **Duplicates Collection**: If we encounter a number whose corresponding index already holds a negative value, it means the number has appeared before, and we add it to the `duplicates` list.

Complexity Analysis:

- **Time Complexity**: O(n), where n is the length of the input array. We make a single pass through the array.
- **Space Complexity**: O(1), as we are using the input array for marking duplicates and only a constant amount of space for the duplicates array.

This solution meets the requirements of running in O(n) time and using constant extra space. If you have any further questions or need clarifications, feel free to ask!

7. 3 Sum

=> To solve the problem of finding all unique triplets in an integer array that sum up to zero, we can use a combination of sorting and the two-pointer technique. This approach helps efficiently find triplets without needing to check every possible combination, thus reducing the time complexity.

Approach:

1. **Sort the Array**: First, sort the input array. This helps in avoiding duplicates and makes it easier to apply the two-pointer technique.
2. **Iterate Through the Array**: For each element in the array, treat it as the first element of the triplet. Then use two pointers to find the other two elements that, together with the first element, sum to zero.
3. **Two-Pointer Technique**:
    - Initialize two pointers: one at the beginning of the remaining elements (after the current element) and the other at the end of the array.
    - Move the pointers towards each other, adjusting them based on whether the sum of the triplet is less than, greater than, or equal to zero.
    - Skip duplicates by moving the pointers appropriately.
4. **Collect Unique Triplets**: Store the triplets in a result array while ensuring there are no duplicates.

Implementation:

Here's how you can implement this in JavaScript:

```
function threeSum(nums) {
    const result = [];
    nums.sort((a, b) => a - b); // Sort the array

    for (let i = 0; i < nums.length - 2; i++) {
        // Skip duplicates for the first element
        if (i > 0 && nums[i] === nums[i - 1]) continue;

        let left = i + 1;
        let right = nums.length - 1;

        while (left < right) {
            const sum = nums[i] + nums[left] + nums[right];

            if (sum === 0) {
                result.push([nums[i], nums[left], nums[right]]);

                // Skip duplicates for the second element
                while (left < right && nums[left] === nums[left + 1]) left++;
                // Skip duplicates for the third element
                while (left < right && nums[right] === nums[right - 1]) right--;

                left++;
                right--;
            } else if (sum < 0) {
                left++; // Move the left pointer to the right to increase the sum
            } else {
                right--; // Move the right pointer to the left to decrease the sum
            }
        }
    }

    return result;
}

// Example usage:
const nums1 = [-1, 0, 1, 2, -1, -4];
console.log(threeSum(nums1)); // Output: [[-1,-1,2],[-1,0,1]]

const nums2 = [0, 1, 1];
console.log(threeSum(nums2)); // Output: []

const nums3 = [0, 0, 0];
console.log(threeSum(nums3)); // Output: [[0,0,0]]
```

Explanation:

1. **Sorting**: The input array is sorted first, which allows us to efficiently find triplets while avoiding duplicates.
2. **Looping Through the Array**: We iterate through each element and use the two-pointer technique to find pairs that sum with the current element to zero.

3. **Duplicate Check**: By skipping duplicates during the iteration, we ensure that the triplets added to the result are unique.
4. **Result Storage**: The valid triplets found are pushed into the result array, which is returned at the end.

Complexity Analysis:

- **Time Complexity**: O(n²), where n is the length of the input array. The outer loop runs in O(n), and for each iteration, the two-pointer search runs in O(n).
- **Space Complexity**: O(1) for the two-pointer approach, but O(k) for storing the result triplets, where k is the number of triplets found.

This approach efficiently finds all unique triplets that sum to zero. If you have any questions or need further explanations, feel free to ask!

8. 4 Sum

=> To solve the problem of finding all unique quadruplets in an integer array that sum up to a given target, we can use a similar approach to the three-sum problem but with an additional loop to accommodate the fourth number. This approach involves sorting the array and using a combination of nested loops and the two-pointer technique to efficiently find the quadruplets.

Approach:

1. **Sort the Array**: This helps in avoiding duplicates and makes it easier to apply the two-pointer technique.
2. **Iterate Through the Array**: Use a nested loop structure where the outer two loops will fix the first two numbers of the quadruplet, and the inner two pointers will find the remaining two numbers that complete the quadruplet to meet the target sum.
3. **Two-Pointer Technique**: For the remaining two numbers, use two pointers to find pairs that sum to the target minus the sum of the two fixed numbers.
4. **Avoid Duplicates**: After finding a quadruplet, skip duplicate values for both fixed and pointer variables to ensure all quadruplets are unique.
5. **Store Results**: Push valid quadruplets to a result array and return it at the end.

Implementation:

Here's how you can implement this in JavaScript:

```
function fourSum(nums, target) {
    const result = [];
    nums.sort((a, b) => a - b); // Sort the array

    const n = nums.length;

    for (let i = 0; i < n - 3; i++) {
        // Skip duplicates for the first element
        if (i > 0 && nums[i] === nums[i - 1]) continue;

        for (let j = i + 1; j < n - 2; j++) {
            // Skip duplicates for the second element
            if (j > i + 1 && nums[j] === nums[j - 1]) continue;

            let left = j + 1;
            let right = n - 1;

            while (left < right) {
                const sum = nums[i] + nums[j] + nums[left] + nums[right];

                if (sum === target) {
                    result.push([nums[i], nums[j], nums[left], nums[right]]);

                    // Skip duplicates for the third element
                    while (left < right && nums[left] === nums[left + 1]) left++;
                    // Skip duplicates for the fourth element
                    while (left < right && nums[right] === nums[right - 1]) right--;

                    left++;
                    right--;
                } else if (sum < target) {
                    left++; // Move the left pointer to the right to increase the sum
                } else {
                    right--; // Move the right pointer to the left to decrease the sum
                }
            }
        }
    }

    return result;
}

// Example usage:
const nums1 = [1, 0, -1, 0, -2, 2];
const target1 = 0;
console.log(fourSum(nums1, target1));
// Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

const nums2 = [2, 2, 2, 2, 2];
const target2 = 8;
```

```
console.log(fourSum(nums2, target2));
// Output: [[2,2,2,2]]
```

Explanation:

1. **Sorting**: The input array is sorted first, which allows us to efficiently find quadruplets while avoiding duplicates.
2. **Nested Loops**: The outer two loops (with indices `i` and `j`) are used to fix the first two numbers. The inner while loop with the two pointers (`left` and `right`) finds the remaining two numbers.
3. **Duplicate Check**: By skipping duplicates during the iterations, we ensure that the quadruplets added to the result are unique.
4. **Result Storage**: The valid quadruplets found are pushed into the result array, which is returned at the end.

Complexity Analysis:

- **Time Complexity**: O(n³), where n is the length of the input array. The outer two loops run in O(n²), and for each pair, the two-pointer search runs in O(n).
- **Space Complexity**: O(1) for the pointers, but O(k) for storing the result quadruplets, where k is the number of quadruplets found.

This approach efficiently finds all unique quadruplets that sum to the target. If you have any further questions or need clarifications, feel free to ask!

9. Search a 2D Matrix

=> To solve the problem of searching for a target integer in a matrix with the properties described, we can treat the matrix as a single sorted array. Since each row is sorted and the first integer of each row is greater than the last integer of the previous row, we can perform a binary search.

Approach:

1. **Understanding the Matrix as a 1D Array**:
    - Given that the matrix is sorted row-wise and that each row's first element is greater than the last element of the previous row, we can imagine the matrix as a single sorted array.
    - The element at position `(i, j)` in the matrix can be accessed as if it were in a 1D array using the formula: `index = i * n + j`, where `n` is the number of columns in the matrix.
2. **Binary Search**:
    - Use binary search to find the target. The search space is defined by the total number of elements in the matrix, which is `m * n`, where `m` is the number of rows and `n` is the number of columns.
    - We will calculate the middle index and map it back to its corresponding row and column in the matrix.

Implementation:

Here's the JavaScript code that implements the binary search approach:

```
function searchMatrix(matrix, target) {
    if (matrix.length === 0 || matrix[0].length === 0) {
        return false; // Handle empty matrix
    }

    const m = matrix.length; // number of rows
    const n = matrix[0].length; // number of columns
    let left = 0;
    let right = m * n - 1; // total number of elements - 1

    while (left <= right) {
        const mid = Math.floor((left + right) / 2);
        const midValue = matrix[Math.floor(mid / n)][mid % n]; // convert mid back to 2D index

        if (midValue === target) {
            return true; // target found
        } else if (midValue < target) {
            left = mid + 1; // move to the right half
        } else {
            right = mid - 1; // move to the left half
        }
    }

    return false; // target not found
}

// Example usage:
const matrix1 = [
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 60]
];
const target1 = 3;
console.log(searchMatrix(matrix1, target1)); // Output: true

const matrix2 = [
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 60]
];
const target2 = 13;
console.log(searchMatrix(matrix2, target2)); // Output: false
```

Explanation:

1. **Input Handling**: The function first checks if the matrix is empty. If it is, it returns `false`.
2. **Binary Search Logic**:
    - We define the search space using `left` and `right` pointers, starting from the beginning to the end of the flattened matrix.
    - The `mid` index is calculated, and we convert it back to the corresponding row and column in the matrix using integer division and modulo operations.

3. **Comparison**:
    - If the value at the `mid` position matches the target, we return `true`.
    - If the value is less than the target, we move the `left` pointer to `mid + 1`.
    - If the value is greater than the target, we move the `right` pointer to `mid - 1`.
4. **Completion**: If the loop finishes without finding the target, we return `false`.

Complexity Analysis:

- **Time Complexity**: O(log(m * n)), as required, because we are effectively performing a binary search on `m * n` elements.
- **Space Complexity**: O(1), since we are using only a few variables for indexing and tracking the search space.

This approach efficiently finds whether the target exists in the matrix, adhering to the specified time complexity constraints. If you have any questions or need further clarification, feel free to ask!

10. Longest Consecutive Sequence

=> To solve the problem of finding the length of the longest consecutive elements sequence in an unsorted array of integers in O(n) time, we can use a HashSet. The idea is to use the HashSet to store all the numbers from the array and then check for the longest consecutive sequence starting from each number.

Approach:

1. **Use a HashSet**: Store all the numbers in a HashSet for O(1) average-time complexity lookups.
2. **Iterate through the array**: For each number, check if it's the start of a sequence (i.e., `num - 1` is not in the set). If it is, count how long the consecutive sequence is by checking for the presence of `num + 1`, `num + 2`, and so on in the HashSet.
3. **Track the maximum length**: Keep track of the maximum length of consecutive sequences found.

Implementation:

Here's the JavaScript code to implement this logic:

```
function longestConsecutive(nums) {
    const numSet = new Set(nums); // Create a HashSet of all numbers
    let maxLength = 0; // To track the maximum length of consecutive sequence

    for (const num of nums) {
        // Check if it's the start of a sequence
        if (!numSet.has(num - 1)) {
            let currentNum = num;
            let currentStreak = 1; // Current streak length

            // Count consecutive numbers
            while (numSet.has(currentNum + 1)) {
                currentNum++;
                currentStreak++;
            }

            // Update max length if current streak is greater
            maxLength = Math.max(maxLength, currentStreak);
        }
    }

    return maxLength; // Return the length of the longest consecutive sequence
}

// Example usage:
const nums1 = [100, 4, 200, 1, 3, 2];
console.log(longestConsecutive(nums1)); // Output: 4

const nums2 = [0, 3, 7, 2, 5, 8, 4, 6, 0, 1];
console.log(longestConsecutive(nums2)); // Output: 9
```

Explanation:

1. **Initialization**:
    - A `Set` is created to store all the numbers from the array for fast lookups.
    - A variable `maxLength` is initialized to keep track of the longest sequence found.
2. **Iteration**:
    - For each number in the input array, we first check if it is the start of a sequence by confirming that `num - 1` is not in the set.
    - If it is the start, we enter a while loop to check how many consecutive numbers follow it (`num + 1`, `num + 2`, etc.), counting them until we reach a number that isn't in the set.
3. **Updating Maximum Length**:
    - After counting the length of the current sequence, we update `maxLength` if the current streak exceeds the previously recorded maximum.
4. **Return Result**:
    - Finally, we return the `maxLength`, which contains the length of the longest consecutive sequence found.

Complexity Analysis:

- **Time Complexity**: O(n), where n is the number of elements in the input array. Each number is processed at most twice (once during the outer loop and once during the inner loop).
- **Space Complexity**: O(n), for storing the numbers in the HashSet.

This approach efficiently finds the longest consecutive sequence while adhering to the O(n) time complexity requirement. If you have any questions or need further clarification, feel free to ask!

11. Find Peak Element

=> To solve the problem of finding a peak element in an array in O(log n) time complexity, we can use a modified binary search approach. The key idea is to exploit the property of peaks in the array. If the middle element is not a peak, then at least one of its neighbors must be greater, allowing us to focus our search on one half of the array.

Steps:

1. **Define a Binary Search**: We will use two pointers, `left` and `right`, to maintain the search range.
2. **Calculate the Middle Index**: At each step, calculate the middle index and check if it is a peak.
3. **Check the Neighbors**: If the middle element is not a peak, determine which side of the array to search next:
    - If the left neighbor is greater, then there must be a peak in the left half (including the left neighbor).
    - If the right neighbor is greater, then there must be a peak in the right half (including the right neighbor).
4. **Repeat Until Found**: Continue this process until a peak is found.

Implementation:

Here's how you can implement this approach in JavaScript:

```
function findPeakElement(nums) {
    let left = 0;
    let right = nums.length - 1;

    while (left < right) {
        const mid = Math.floor((left + right) / 2);

        // Compare the middle element with its right neighbor
        if (nums[mid] < nums[mid + 1]) {
            // If the right neighbor is greater, search in the right half
            left = mid + 1;
        } else {
            // If the middle element is greater or equal, search in the left half
            right = mid;
        }
    }

    // At the end of the loop, left == right, which is the peak's index
    return left;
}

// Example usage:
const nums1 = [1, 2, 3, 1];
console.log(findPeakElement(nums1)); // Output: 2

const nums2 = [1, 2, 1, 3, 5, 6, 4];
console.log(findPeakElement(nums2)); // Output: 5
```

Explanation:

1. **Initialization**: We set `left` to 0 and `right` to the last index of the array.
2. **While Loop**: The loop runs as long as `left` is less than `right`.
    - We calculate the `mid` index.
    - We compare `nums[mid]` with `nums[mid + 1]`:
        - If `nums[mid] < nums[mid + 1]`, it indicates that there is a peak in the right half of the array, so we adjust `left` to `mid + 1`.
        - If `nums[mid] >= nums[mid + 1]`, it suggests a peak might exist in the left half, so we adjust `right` to `mid`.
3. **Return**: When `left` equals `right`, we return `left`, which is the index of a peak element.

Complexity Analysis:

- **Time Complexity**: O(log n) because we are halving the search space at each step.
- **Space Complexity**: O(1) since we are using a constant amount of extra space.

This method efficiently finds a peak element index in logarithmic time. Let me know if you have any questions or need further assistance!

12. Single Element in a Sorted Array

=> To find the single element in a sorted array where every other element appears exactly twice, we can use a binary search approach. The key insight is that the positions of the elements will help us determine which half of the array to search.

Key Observations:

1. In a sorted array where every element appears twice except for one, if we look at the index of the elements, we can observe:
    - If the single element is to the left of a pair, then all elements to the left of it will be in positions where the index is even.
    - If the single element is to the right of a pair, then all elements to the left of it will be in positions where the index is odd.
2. The property of pairs in the array allows us to adjust our search based on the parity (even or odd) of the mid index.

Steps:

1. **Initialize pointers**: Use `left` and `right` to represent the bounds of our search space.
2. **Binary Search**: Calculate the mid-point:
    - If `mid` is even and `nums[mid]` is equal to `nums[mid + 1]`, it means the single element is to the right.
    - If `mid` is even and `nums[mid]` is not equal to `nums[mid + 1]`, the single element is on the left including mid.
    - If `mid` is odd, reverse the conditions.
3. **Narrow down the search** until we find the single element.

Implementation:

Here's the implementation in JavaScript:

```
function singleNonDuplicate(nums) {
    let left = 0;
    let right = nums.length - 1;

    while (left < right) {
        const mid = Math.floor((left + right) / 2);

        // Ensure mid is even
        if (mid % 2 === 1) {
            mid--; // Make it even to compare with the next element
        }

        // Compare pairs
        if (nums[mid] === nums[mid + 1]) {
            // The single element must be in the right half
            left = mid + 2;
        } else {
            // The single element is in the left half
            right = mid;
        }
    }

    // left will point to the single element
    return nums[left];
}

// Example usage:
const nums1 = [1, 1, 2, 3, 3, 4, 4, 8, 8];
console.log(singleNonDuplicate(nums1)); // Output: 2

const nums2 = [3, 3, 7, 7, 10, 11, 11];
console.log(singleNonDuplicate(nums2)); // Output: 10
```

Explanation:

1. **Initialization**: `left` starts at 0 and `right` at the last index of the array.
2. **While Loop**: The loop continues until `left` equals `right`.
    - Calculate `mid` and ensure it is even for pair comparison.
    - Depending on the comparison of `nums[mid]` and `nums[mid + 1]`, update `left` or `right` to narrow down the search.
3. **Return the Result**: After exiting the loop, `left` will be at the index of the single element.

Complexity Analysis:

- **Time Complexity**: O(log n) because we are halving the search space with each iteration.
- **Space Complexity**: O(1) since we are using a constant amount of space.

This method efficiently finds the single non-duplicate element in a sorted array. Let me know if you have any questions or need further assistance!

13. Find Minimum in Rotated Sorted Array

=> To find the minimum element in a sorted and rotated array in O(log n) time, we can use a modified binary search algorithm. The key observation is that even though the array is rotated, one half of the array will always be sorted.

Steps:

1. **Initialize pointers**: Use two pointers, `left` and `right`, to represent the bounds of the search space.
2. **Binary Search**: While `left` is less than `right`:
    - Calculate the mid-point.
    - Compare the middle element with the rightmost element:
        - If `nums[mid]` is greater than `nums[right]`, it means the minimum element must be in the right half (because the right half is unsorted).
        - If `nums[mid]` is less than or equal to `nums[right]`, it means the minimum element must be in the left half (because the left half is sorted).
3. **Narrow down the search** until `left` equals `right`, which will be the minimum element.

Implementation:

Here's how you can implement this in JavaScript:

```
function findMin(nums) {
    let left = 0;
    let right = nums.length - 1;

    while (left < right) {
        const mid = Math.floor((left + right) / 2);

        // Compare mid element with the rightmost element
        if (nums[mid] > nums[right]) {
            // Minimum must be in the right half
            left = mid + 1;
        } else {
            // Minimum is in the left half including mid
            right = mid;
        }
    }

    // left will be pointing to the minimum element
    return nums[left];
}

// Example usage:
const nums1 = [3, 4, 5, 1, 2];
console.log(findMin(nums1)); // Output: 1

const nums2 = [4, 5, 6, 7, 0, 1, 2];
console.log(findMin(nums2)); // Output: 0

const nums3 = [11, 13, 15, 17];
console.log(findMin(nums3)); // Output: 11
```

Explanation:

1. **Initialization**: We start with `left` at index 0 and `right` at the last index of the array.
2. **While Loop**: We perform binary search until `left` is less than `right`.
    - Calculate `mid`.
    - If the middle element is greater than the rightmost element, this means the smallest value is in the right portion, so we move `left` to `mid + 1`.
    - If the middle element is less than or equal to the rightmost element, this means the smallest value is in the left portion (including the middle), so we set `right` to `mid`.
3. **Return Result**: After the loop exits, `left` will be pointing to the minimum element in the array.

Complexity Analysis:

- **Time Complexity**: O(log n) since we are halving the search space in each iteration.
- **Space Complexity**: O(1) because we are using a constant amount of space.

This method efficiently finds the minimum element in a rotated sorted array. Let me know if you have any questions or need further explanations!

14. Search in Rotated Sorted Array II

=> To determine if a target value exists in a rotated sorted array that may contain duplicates, we can use a modified binary search algorithm. The key here is to handle the duplicates effectively while still maintaining a logarithmic time complexity as much as possible.

Approach:

1. **Initialization**: Set two pointers, `left` and `right`, to the start and end of the array.
2. **Binary Search**: While `left` is less than or equal to `right`:
    - Calculate the mid-point.
    - If the middle element equals the target, return `true`.
    - If the left element equals the middle element, increment `left` by one to skip duplicates.
    - Determine which half is sorted:
        - If the left half is sorted (`nums[left] <= nums[mid]`):
            - Check if the target is in the sorted half:
                - If `nums[left] <= target < nums[mid]`, adjust `right` to `mid - 1`.
                - Otherwise, move `left` to `mid + 1`.
        - If the right half is sorted (`nums[mid] <= nums[right]`):
            - Check if the target is in the sorted half:
                - If `nums[mid] < target <= nums[right]`, adjust `left` to `mid + 1`.
                - Otherwise, move `right` to `mid - 1`.
3. If the loop exits without finding the target, return `false`.

Implementation:

Here's how you can implement this in JavaScript:

```
function search(nums, target) {
    let left = 0;
    let right = nums.length - 1;

    while (left <= right) {
        const mid = Math.floor((left + right) / 2);

        // Check if the middle element is the target
        if (nums[mid] === target) {
            return true;
        }

        // If duplicates are encountered, skip them
        if (nums[left] === nums[mid]) {
            left++; // Move left pointer to the right
        } else if (nums[left] <= nums[mid]) { // Left half is sorted
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1; // Target is in the left half
            } else {
                left = mid + 1; // Target is in the right half
            }
        } else { // Right half is sorted
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1; // Target is in the right half
            } else {
                right = mid - 1; // Target is in the left half
            }
        }
    }

    return false; // Target not found
}

// Example usage:
const nums1 = [2, 5, 6, 0, 0, 1, 2];
const target1 = 0;
console.log(search(nums1, target1)); // Output: true

const nums2 = [2, 5, 6, 0, 0, 1, 2];
const target2 = 3;
console.log(search(nums2, target2)); // Output: false
```

Explanation of the Code:

1. **Initialization**: The `left` pointer starts at 0 and the `right` pointer starts at the last index of the array.
2. **Loop**: While `left` is less than or equal to `right`, we calculate the mid-point and check for the target.
3. **Handling Duplicates**: If we encounter duplicates at the left and mid, we simply move the left pointer right by one to bypass the duplicate value.
4. **Determining the Sorted Half**: We check which half is sorted and adjust our search space accordingly based on whether the target could lie in the sorted half.

5. **Result**: If the target is found, we return `true`; otherwise, if the loop completes without finding the target, we return `false`.

Complexity Analysis:

- **Time Complexity**: O(log n) in average case, but can degrade to O(n) in the worst case due to duplicates.
- **Space Complexity**: O(1) since we are using a constant amount of space.

This method efficiently checks for the existence of the target in a rotated sorted array, even when duplicates are present. Let me know if you have any questions or need further clarification!

15. Search in Rotated Sorted Array

=> To find the index of a target value in a rotated sorted array, we can use a modified binary search algorithm. The algorithm is efficient, with a time complexity of (O(\log n)).

Approach:

1. **Initialization**: Set two pointers, `left` and `right`, to the start and end of the array.
2. **Binary Search**: While `left` is less than or equal to `right`:
    - Calculate the mid-point.
    - Check if the middle element is equal to the target. If it is, return the mid index.
    - Determine which half of the array is sorted:
        - If the left half is sorted:
            - Check if the target is in the left half. If it is, adjust the `right` pointer to `mid - 1`. If not, move the `left` pointer to `mid + 1`.
        - If the right half is sorted:
            - Check if the target is in the right half. If it is, adjust the `left` pointer to `mid + 1`. If not, move the `right` pointer to `mid - 1`.
3. If the loop exits without finding the target, return -1.

Implementation:

Here’s the implementation in JavaScript:

```
function search(nums, target) {
    let left = 0;
    let right = nums.length - 1;

    while (left <= right) {
        const mid = Math.floor((left + right) / 2);

        // Check if the middle element is the target
        if (nums[mid] === target) {
            return mid;
        }

        // Determine which half is sorted
        if (nums[left] <= nums[mid]) { // Left half is sorted
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1; // Target is in the left half
            } else {
                left = mid + 1; // Target is in the right half
            }
        } else { // Right half is sorted
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1; // Target is in the right half
            } else {
                right = mid - 1; // Target is in the left half
            }
        }
    }

    return -1; // Target not found
}

// Example usage:
const nums1 = [4, 5, 6, 7, 0, 1, 2];
const target1 = 0;
console.log(search(nums1, target1)); // Output: 4

const nums2 = [4, 5, 6, 7, 0, 1, 2];
const target2 = 3;
console.log(search(nums2, target2)); // Output: -1
```

Explanation of the Code:

1. **Initialization**: The `left` pointer starts at the beginning of the array, and the `right` pointer starts at the end.
2. **Loop**: As long as `left` is less than or equal to `right`, we calculate the mid-point and check if the mid element matches the target.
3. **Identifying Sorted Halves**:
    - If the left part is sorted (`nums[left] <= nums[mid]`), we check if the target lies within this part and adjust pointers accordingly.
    - If the right part is sorted, we do the same check and adjustment for the right half.
4. **Return Result**: If the target is found, return its index. If the search completes without finding the target, return -1.

Complexity Analysis:

- **Time Complexity**: (O(\log n)) due to the binary search approach.
- **Space Complexity**: (O(1)) as we are using a constant amount of space.

This approach is efficient and well-suited for searching in a rotated sorted array. Let me know if you have any questions or need further clarification!

16. Find First and Last Position of Element in Sorted Array

=> To find the starting and ending positions of a given target value in a sorted array, we can use a modified binary search approach. The goal is to find the first and last positions of the target efficiently with a time complexity of (O(\log n)).

Approach:

1. **Binary Search for First Occurrence**: Perform a binary search to find the first occurrence of the target.
2. **Binary Search for Last Occurrence**: Perform another binary search to find the last occurrence of the target.
3. If the target is found, return the positions. If not, return `[-1, -1]`.

Implementation:

Here's the implementation in JavaScript:

```
function searchRange(nums, target) {
    const findFirst = (nums, target) => {
        let left = 0;
        let right = nums.length - 1;
        let firstPosition = -1;

        while (left <= right) {
            const mid = Math.floor((left + right) / 2);

            if (nums[mid] === target) {
                firstPosition = mid; // Update firstPosition
                right = mid - 1; // Look for an earlier occurrence
            } else if (nums[mid] < target) {
                left = mid + 1; // Move right
            } else {
                right = mid - 1; // Move left
            }
        }

        return firstPosition;
    };

    const findLast = (nums, target) => {
        let left = 0;
        let right = nums.length - 1;
        let lastPosition = -1;

        while (left <= right) {
            const mid = Math.floor((left + right) / 2);

            if (nums[mid] === target) {
                lastPosition = mid; // Update lastPosition
                left = mid + 1; // Look for a later occurrence
            } else if (nums[mid] < target) {
                left = mid + 1; // Move right
            } else {
                right = mid - 1; // Move left
            }
        }

        return lastPosition;
    };

    const start = findFirst(nums, target);
    const end = findLast(nums, target);

    return start === -1 ? [-1, -1] : [start, end];
}

// Example usage:
const nums1 = [5, 7, 7, 8, 8, 10];
```

```
const target1 = 8;
console.log(searchRange(nums1, target1)); // Output: [3, 4]

const nums2 = [5, 7, 7, 8, 8, 10];
const target2 = 6;
console.log(searchRange(nums2, target2)); // Output: [-1, -1]
```

Explanation of the Code:

1. **Function `findFirst`**:
    - This function performs binary search to find the first occurrence of the target. When the target is found, it updates `firstPosition` and continues to search in the left half of the array to find earlier occurrences.
2. **Function `findLast`**:
    - This function performs binary search to find the last occurrence of the target. When the target is found, it updates `lastPosition` and continues to search in the right half of the array to find later occurrences.
3. **Main Function**:
    - The main function `searchRange` calls `findFirst` and `findLast` to get the starting and ending positions. If the starting position is `-1`, it indicates that the target is not present in the array, so it returns `[-1, -1]`. Otherwise, it returns the positions.

Complexity Analysis:

- **Time Complexity**: (O(\log n)) for each of the two binary searches, leading to a total of (O(\log n)).
- **Space Complexity**: (O(1)) since we are using a constant amount of extra space.

This approach efficiently finds the required positions in a sorted array. Let me know if you have any questions or need further clarification!

17. Search a 2D Matrix II

=> To efficiently search for a target value in a 2D matrix with the properties that each row is sorted in ascending order from left to right and each column is sorted in ascending order from top to bottom, we can utilize a zigzag search approach starting from the top-right corner of the matrix.

Approach:

1. **Start from the Top-Right Corner**:
    - If the current element is equal to the target, return `true`.
    - If the current element is greater than the target, move left to the next column (decrease the column index).
    - If the current element is less than the target, move down to the next row (increase the row index).
2. **Continue Until the Search Space is Exhausted**: If you go out of bounds of the matrix, return `false`.

Implementation:

Here's the implementation in JavaScript:

```
function searchMatrix(matrix, target) {
    if (matrix.length === 0 || matrix[0].length === 0) {
        return false; // Early exit for empty matrix
    }

    const rows = matrix.length;
    const cols = matrix[0].length;
    let row = 0; // Start from the first row
    let col = cols - 1; // Start from the last column

    while (row < rows && col >= 0) {
        const currentElement = matrix[row][col];

        if (currentElement === target) {
            return true; // Target found
        } else if (currentElement > target) {
            col--; // Move left
        } else {
            row++; // Move down
        }
    }

    return false; // Target not found
}

// Example usage:
const matrix1 = [
    [1, 4, 7, 11, 15],
    [2, 5, 8, 12, 19],
    [3, 6, 9, 16, 22],
    [10, 13, 14, 17, 24],
    [18, 21, 23, 26, 30]
];
const target1 = 5;
console.log(searchMatrix(matrix1, target1)); // Output: true

const target2 = 20;
console.log(searchMatrix(matrix1, target2)); // Output: false
```

Explanation of the Code:

1. **Initial Checks**: We check if the matrix is empty. If it is, we return `false`.
2. **Variables**: We define `rows` and `cols` to store the number of rows and columns in the matrix. We start from the top-right corner by initializing `row` to `0` and `col` to `cols - 1`.
3. **Loop**: We continue to loop while the `row` is within bounds and `col` is non-negative:
    - **Current Element Check**: We check the value of the current element.
    - If it equals the target, we return `true`.
    - If it's greater, we move left (decrease `col`).
    - If it's less, we move down (increase `row`).
4. **Return**: If the loop ends without finding the target, we return `false`.

Complexity Analysis:

- **Time Complexity**: (O(m + n)), where (m) is the number of rows and (n) is the number of columns. In the worst case, we may traverse the entire height and width of the matrix.
- **Space Complexity**: (O(1)) since we are using a constant amount of space.

This approach effectively utilizes the properties of the sorted matrix to efficiently search for the target. If you have any further questions or need additional explanations, feel free to ask!

18. Find a Peak Element II

=> To find a peak element in a 2D grid with the properties described, we can utilize a modified binary search approach that efficiently narrows down the search space. Here's how we can implement this algorithm:

Approach:

1. **Binary Search in Columns**: Choose the middle column and find the global maximum in that column. This element has the potential to be a peak.
2. **Check Neighbors**: Compare the selected maximum element with its neighbors (up, down, left, right).
    - If it is greater than its neighbors, we have found a peak, and we can return its position.
    - If it is less than the neighbor above, we move to the upper half of the current column.
    - If it is less than the neighbor below, we move to the lower half of the current column.
3. **Repeat**: This process continues until we find a peak.

Implementation:

Here's the implementation in Python:

```
def findPeakGrid(mat):
    if not mat or not mat[0]:
        return [-1, -1]  # Handle edge case for empty matrix

    m, n = len(mat), len(mat[0])

    left, right = 0, n - 1

    while left < right:
        mid_col = (left + right) // 2

        # Find the maximum element in the mid_col
        max_row = 0
        for i in range(m):
            if mat[i][mid_col] > mat[max_row][mid_col]:
                max_row = i

        # Compare the maximum element with its neighbors
        # Check if it's greater than the right neighbor
        if mid_col + 1 < n and mat[max_row][mid_col] < mat[max_row][mid_col + 1]:
            left = mid_col + 1  # Move to the right side
        else:
            # If it's not smaller than the right neighbor, we check the left neighbor
            right = mid_col  # Move to the left side

    # After narrowing down, we find the maximum in the last narrowed column
    max_row = 0
    for i in range(m):
        if mat[i][left] > mat[max_row][left]:
            max_row = i

    return [max_row, left]

# Example usage
mat1 = [[1, 4], [3, 2]]
print(findPeakGrid(mat1))  # Output: [0, 1]
```

Explanation of the Code:

1. **Initialization**: We determine the dimensions of the matrix (`m` for rows and `n` for columns). We set our binary search bounds with `left` and `right`.
2. **Binary Search Loop**: We perform the binary search in the columns until `left` is equal to `right`:
   - We calculate the middle column `mid_col`.
   - We find the maximum element's row index in the current `mid_col`.
3. **Comparison**:
   - If the maximum element is less than its right neighbor, we narrow our search to the right half (moving `left` to `mid_col + 1`).
   - If it's greater than or equal to its right neighbor, we narrow our search to the left half (moving `right` to `mid_col`).
4. **Final Check**: After determining the narrow column, we find the maximum element in that column, which is guaranteed to be a peak.
5. **Return**: Finally, we return the coordinates of the peak.

Complexity Analysis:

- **Time Complexity**: The algorithm runs in (O(m \log(n))) or (O(n \log(m))) because for each column selection, we scan the rows to find the maximum.
- **Space Complexity**: The space complexity is (O(1)) as we are using a constant amount of space.

This approach efficiently finds a peak element in the 2D grid while adhering to the constraints provided. If you have any further questions or need more explanations, feel free to ask!

19. Search Insert Position

=> To solve the problem of finding the index of a target value in a sorted array or determining where it should be inserted, we can use a binary search approach. The binary search will allow us to achieve the required O(log n) runtime complexity.

Approach:

1. **Initialize Pointers**: We will maintain two pointers, `left` and `right`, to represent the current search bounds within the array.
2. **Binary Search Loop**: We will continue to adjust the bounds based on the middle element:
    - If the middle element equals the target, we return the index of that element.
    - If the middle element is less than the target, we move the `left` pointer to `mid + 1`.
    - If the middle element is greater than the target, we move the `right` pointer to `mid - 1`.
3. **Insertion Point**: If the target is not found, the loop will terminate when `left` exceeds `right`. The `left` pointer will then represent the index where the target should be inserted.

Implementation:

Here's the implementation in Python:

```
def searchInsert(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = (left + right) // 2

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    # If we exit the loop, left is the index to insert the target
    return left

# Example usage
nums1 = [1, 3, 5, 6]
target1 = 5
print(searchInsert(nums1, target1))  # Output: 2

nums2 = [1, 3, 5, 6]
target2 = 2
print(searchInsert(nums2, target2))  # Output: 1
```

Explanation of the Code:

1. **Initialization**: We set `left` to the start of the array (0) and `right` to the end of the array (length - 1).
2. **Binary Search Loop**:
   - We calculate the middle index `mid` as the average of `left` and `right`.
   - We check if the element at `mid` is equal to the target. If so, we return `mid`.
   - If the element at `mid` is less than the target, we adjust `left` to `mid + 1` to search in the right half.
   - If the element at `mid` is greater than the target, we adjust `right` to `mid - 1` to search in the left half.
3. **Insertion Point**: When the loop exits, it means the target was not found. At this point, `left` points to the correct insertion index for the target.

Complexity Analysis:

- **Time Complexity**: The algorithm runs in (O(\log n)) due to the binary search mechanism.
- **Space Complexity**: The space complexity is (O(1)) since we are using a constant amount of space for pointers.

This implementation effectively finds the target's index or the correct insertion position in a sorted array. Let me know if you need any further assistance or explanations!

20. Binary Search

=> To search for a target value in a sorted array with a runtime complexity of (O(\log n)), we can implement a binary search algorithm. Here's how it works:

Steps of the Binary Search Algorithm:

1. **Initialize Pointers**: Start with two pointers, `left` at the beginning of the array and `right` at the end of the array.
2. **Iterate Until Found or Exhausted**:
    - Calculate the middle index `mid` as the average of `left` and `right`.
    - Compare the value at the `mid` index with the target:
        - If they are equal, return `mid`.
        - If the value at `mid` is less than the target, move the `left` pointer to `mid + 1` (search in the right half).
        - If the value at `mid` is greater than the target, move the `right` pointer to `mid - 1` (search in the left half).
3. **Return -1**: If the target is not found after the loop exits, return -1.

Implementation:

Here is the Python implementation of the binary search algorithm:

```
def search(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = (left + right) // 2

        # Check if the target is present at mid
        if nums[mid] == target:
            return mid
        # If target is greater, ignore the left half
        elif nums[mid] < target:
            left = mid + 1
        # If target is smaller, ignore the right half
        else:
            right = mid - 1

    # Target not found
    return -1

# Example usage
nums1 = [-1, 0, 3, 5, 9, 12]
target1 = 9
print(search(nums1, target1))  # Output: 4

nums2 = [-1, 0, 3, 5, 9, 12]
target2 = 2
print(search(nums2, target2))  # Output: -1
```

Explanation of the Code:

- **Initialization**: We set `left` to 0 (the first index) and `right` to `len(nums) - 1` (the last index).
- **Binary Search Loop**:
    - We continuously check the mid-point until the `left` pointer exceeds the `right` pointer.
    - If the middle element equals the target, we return the index `mid`.
    - Depending on whether the middle element is less than or greater than the target, we adjust the `left` or `right` pointers accordingly.
- **Returning -1**: If we exit the loop without finding the target, we return -1.

Complexity Analysis:

- **Time Complexity**: The time complexity is (O(\log n)) because we halve the search space with each iteration.
- **Space Complexity**: The space complexity is (O(1)) as we use a constant amount of extra space.

This implementation effectively finds the index of the target in a sorted array or indicates that it is not present. Let me know if you have any questions or need further clarifications!