

## **1. JavaScript & DSA with 700+ Questions Interview:**

### What's Included?

- ✓ **Theory Questions:** Covers both **basic & advanced** JavaScript topics in a simplified way.
- ✓ **Output-Based Questions:** Tests how well you understand **JavaScript behavior** with real examples.
- ✓ **Coding Tasks:** Practical problems with **step-by-step solutions** to build confidence.

### **DSA & System Design Questions:**

- ✓ **550+ LeetCode-style coding problems** from **FAANG & Hackathons**
  - ✓ Covers **Arrays, Linked Lists, Trees, Graphs, Dynamic Programming, Sorting & Searching**
  - ✓ **Updated weekly interview questions**
- 

### Topics Covered in JavaScript:

- Introduction to JavaScript
- Type Coercion
- Data Types & Operators
- Conditional Statements, Switch & Loops
- Arrays & Array Methods
- Functions & Closures
- Objects & Prototypes
- DOM (Document Object Model)
- Event Handlers, Event Propagation & Delegation
- Event Loop & Asynchronous JavaScript
- Spread Operator, Rest Parameter & Destructuring
- Classes & Inheritance
- Arrow Functions & Callbacks
- Higher-Order Functions (HOF), map, filter, reduce
- Debouncing & Throttling
- Modules, Cookies, Local Storage, Session Storage

### DSA Syllabus: 550+ Coding Questions

#### **1. Arrays & List Processing**

- Pair Matching in Arrays
- Stock Price Optimization
- Contains Duplicate
- Array Product Computation
- Maximum Subarray & Maximum Product Subarray
- Optimizing Array Search
- Multi-Sum Combinations in Arrays

- Water Collection in Structures

## 2. Bit Manipulation & Binary Operations

- Sum Calculation Without Arithmetic Operators
- Number of 1 Bits
- Counting Bits
- Missing Number

## 3. Interval & Range-Based Problems

- Efficient Range Insertions in an Array
- Merge Intervals
- Non-overlapping Intervals

## 4. Linked List Operations

- Reversing Elements in a Linked List
- Detecting Cycles in Linked Lists
- Merging Two Ordered Linked Lists
- Merge K Sorted Linked Lists
- Removing Specific Nodes in a Sequence
- Optimizing Linked List Reordering

## 5. Matrix & Grid-Based Challenges

- Setting Zero Values in Matrices Efficiently
- Traversing a Spiral Matrix

## 6. String Manipulation & Pattern Matching

- Longest Substring Without Repeating Characters
- Detecting Repeated Character Patterns
- Minimum Window Substring
- Valid Parentheses
- Valid Palindrome
- Longest Palindromic Substring
- Palindromic Substrings
- Encoding & Decoding Text-Based Data

## 7. Trees & Hierarchical Data Structures

- Determining Tree Depth & Size
- Comparing Two Tree Structures
- Invert/Flip Binary Tree
- Finding the Maximum Path in Trees
- Level-Wise Traversal of Trees
- Serializing & Restoring Tree Data
- Checking Subtree Presence
- Constructing Trees from Order Sequences

- Validate Binary Search Tree
- Kth Smallest Element in a BST
- Optimizing Word Storage with Prefix Trees
- Advanced Word Search in Trees
- Word Search II

## 8. Graphs & Connectivity Problems

- Cloning Graph Structures
- Planning Task Dependencies
- Tracking Water Flow Across Networks
- Finding the Longest Connected Sequences

## 9. Dynamic Programming & Optimization Strategies

- Staircase Climbing with Variable Steps
- Optimizing Coin Exchange Strategies
- Longest Increasing Subsequence
- Longest Common Subsequence
- Breaking Words into Meaningful Segments
- Finding the Best Combination Selections
- House Robber, House Robber II
- Unique Paths
- Predicting Jump Patterns in Arrays

## 10. Heaps & Searching, Sorting

- Merging Multiple Sorted Data Sets
  - Finding the Top K Frequent Elements
  - Handling Streaming Data for Median Computation
-

## Introduction to JavaScript:

### **1. What are JavaScript engines, and why are they important?**

**Answer:**

1. JavaScript engines are programs that execute JavaScript code.
2. Examples of JavaScript engines:
  - o **V8 Engine**: Used in Chrome and Node.js.
  - o **SpiderMonkey**: Used in Firefox.
3. They optimize JavaScript code for faster execution.
4. Enable modern web capabilities by enhancing performance and speed.
5. Critical for running JavaScript applications in both browsers and server environments.

### **2. How is JavaScript executed in the browser?**

**Answer:**

1. The browser's JavaScript engine (e.g., V8 for Chrome) reads and interprets JavaScript code.
2. Converts JavaScript into machine code using techniques like **Just-In-Time (JIT) compilation**.
3. Executes the machine code to produce dynamic and interactive results on the webpage.
4. Handles updates to the DOM and browser APIs efficiently.
5. Enables smooth interactivity and responsiveness in modern web applications.

### **3. What are the limitations of JavaScript?**

**Answer:**

1. **Security**: Can be exploited for malicious purposes if not handled correctly.
2. **Browser Dependency**: JavaScript behavior may vary slightly across different browsers.
3. **No Multithreading**: Runs on a single thread, which can limit performance for heavy computations.
4. **Sandboxed Environment**: Cannot access system-level resources directly.
5. **Dynamic Typing**: While flexible, it can lead to runtime errors if types are misused.

### **4. How do you include JavaScript in an HTML file?**

**Answer:**

1. **Inline**: Define JavaScript code directly within an HTML element.

Example:

```
<button onclick="alert('Hello!')">Click Me</button>;
```

2. **Internal**: Use a <script> tag within the same HTML file.

Example:

```
<script>console.log('Hello');</script>;
```

3. **External:** Link an external JavaScript file using the `<script>` tag with the `src` attribute.

Example:

```
<script src="script.js"></script>;
```

4. Inline scripts are ideal for small tasks, while external scripts improve maintainability for larger applications.
5. Internal scripts allow defining custom logic within the same file without external dependencies.

## 5. Why is JavaScript considered single-threaded?

**Answer:**

1. JavaScript uses a **single-threaded event loop** to execute code sequentially.
2. This design avoids the complexities and potential bugs of multithreading.
3. Handles **asynchronous operations** like API calls using callbacks, promises, and `async/await`.
4. Efficiently processes tasks via an event loop and a callback queue.
5. Ensures simplicity and predictability in application behavior.

## 6. Is JavaScript a Dynamically Typed and Why?

**Answer:**

1. Yes, JavaScript is dynamically typed, meaning variable types are determined at runtime.
2. Variables do not require explicit type declarations during initialization.
3. The type of a variable can change dynamically during execution.
4. Allows flexibility and rapid prototyping but increases the risk of runtime errors if types are misused.

Example:

### Example of Dynamic Typing in JavaScript:

```
// Declaring a variable without specifying a type
let data;
// Assigning a number to the variable
data = 42;
console.log(typeof data); // Output: "number"
// Reassigning a string to the same variable
data = "Hello, world!";
console.log(typeof data); // Output: "string"
// Reassigning a boolean to the same variable
```

```
data = true;  
console.log(typeof data); // Output: "boolean"
```

## Why is JavaScript Dynamically Typed?

### Answer:

1. Variables do not have a fixed type.
2. The type of a variable is determined by the value assigned to it.
3. This flexibility allows for rapid prototyping but can lead to runtime errors if types are misused.

## Comparison with Statically Typed Languages:

In statically typed languages like **Java**, you must define the variable's type at declaration, and it cannot change later.

### Example in Java (Statically Typed Language):

```
int number = 42; // Variable type is explicitly declared  
number = "Hello"; // Error: Incompatible types
```

## Data Types :

### 7. What is the difference between primitive and non-primitive data types?

### Answer:

Feature	Primitive Types	Non-Primitive Types
Mutability	Immutable (values cannot be changed)	Mutable (can change values)
Storage	Stored directly in the variable	Stored as a reference in memory
Examples	Number, String, Boolean, Undefined, Null, Symbol, BigInt	Object, Array, Function

### 8. What is the difference between undefined and null?

### Answer:

### Undefined:

1. A variable is declared but not assigned a value.
2. Indicates a variable is not initialized.

```
let x;
```

```
console.log(x); // Output: undefined
```

**Null:**

1. Represents the intentional absence of any value.
2. It is explicitly assigned.

```
let y = null;  
console.log(y); // Output: null
```

**9. How does JavaScript handle type coercion?**

**Answer:**

- Automatically converts one data type to another when required.
- Coercion depends on the operation being performed.

```
console.log('5' - 2); // Output: 3 (string '5' is coerced to a number)  
console.log('5' + 2); // Output: '52' (number 2 is coerced to a string)
```

- Coercion can happen with operators like +, -, ==, etc.

**10. What are Symbol and BigInt in JavaScript?**

**Answer:**

**Symbol:**

- Introduced in ES6.
- Represents a unique, immutable identifier.
- Commonly used as object keys to avoid property name conflicts.

```
const sym1 = Symbol('unique');  
const sym2 = Symbol('unique');  
console.log(sym1 === sym2); // Output: false
```

**BigInt:**

- Introduced in ES11 (ES2020).
- Used for numbers larger than  $2^{53} - 1$ , which cannot be safely represented using Number.

Example:

```
const bigNum = 123456789012345678901234567890n;  
console.log(bigNum); // Output: 123456789012345678901234567890n
```

**13. What is NaN in JavaScript?**

**Answer:**

- **NaN (Not-a-Number):** A special numeric value indicating a computation result that isn't a valid number.
- It is of type Number.

```
console.log(typeof NaN); // Output: number
console.log(0 / 0); // Output: NaN
```

## 14. How are objects and arrays different in JavaScript?

**Answer:**

**Object:**

- Stores key-value pairs.
- Keys can be strings or symbols.

```
const obj = { name: "John", age: 30 };
```

**Array:**

- Stores ordered data (indexed by numbers).

```
const arr = [1, 2, 3, 4];
```

**Operators:**

## 15. How does the ternary operator work?

**Answer:** The ternary operator is a shorthand for an if-else statement. The syntax is:

```
condition ? expression1 : expression2;
```

Example:

```
const age = 18;
const canVote = age >= 18 ? "Yes" : "No";
console.log(canVote); // Output: Yes
```

## 16. What is the difference between null and undefined with the == and === operators?

**Answer:**

- **== (Abstract Equality):** Compares values after type coercion.

```
console.log('5' == 5); // Output: true
```

**=== (Strict Equality):** Compares values **and types** without type coercion.

```
console.log('5' === 5); // Output: false
```

null and undefined with the == and === operators

```
console.log(null == undefined); // Output: true
```

The values are treated as equivalent

```
console.log(null === undefined); // Output: false
```

Strict comparison considers their types different.

## 17. Explain the typeof operator with examples.

**Answer:** The typeof operator returns the type of a variable.

Examples:

```
console.log(typeof 42); // Output: number
console.log(typeof "Hello"); // Output: string
console.log(typeof true); // Output: boolean
console.log(typeof undefined); // Output: undefined
console.log(typeof null); // Output: object (a legacy bug in JavaScript)
console.log(typeof []); // Output: object
console.log(typeof function(){}); // Output: function
```

## 18. What is the difference between ?? and || operator in JavaScript?

**Answer:**

**??:**

Specifically designed for scenarios where you want to handle **null or undefined values only** and leave other falsy values untouched.

**Why use ??:**

- Prevents overriding valid falsy values like 0, false, or "".
- Makes code more explicit when dealing with nullish values.

```
let result = value1 ?? value2;
```

If value1 is **not null or undefined**, the result is value1. If value1 is **null or undefined**, the result is value2.

Example:

```
let username = null;
let defaultUsername = "Guest";

let finalUsername = username ?? defaultUsername;

console.log(finalUsername); // Output: "Guest"
```

username is null, so defaultUsername is used.

**||:**

Designed for cases where you want to handle **all falsy values**, not just null or undefined. This is useful when any falsy value represents an invalid input and needs a fallback.

**Why use ||:**

- Provides a general-purpose fallback mechanism for any falsy value.
- Common in scenarios like defaulting an empty string, 0, or false to another value.

**Example:**

```
let age = 0; // User input, 0 is falsy  
console.log(age || 18); // Output: 18 (0 is treated as invalid)
```

## 19. What is the difference between `++x` and `x++`?

**Answer:**

- **++x (Pre-increment):** Increments the value of x and returns the updated value.

```
let x = 5;  
console.log(++x); // Output: 6
```

- **x++ (Post-increment):** Returns the current value of x and then increments it.

```
let x = 5;  
console.log(x++); // Output: 5  
console.log(x); // Output: 6
```

**control flow:**

## 20. What are the types of control flow statements in JavaScript?

**Answer:**

1. **Conditional Statements:**
  - if, else if, else
  - switch
2. **Loops:**
  - for, while, do...while, for...in, for...of
3. **Jump Statements:**
  - break, continue
4. **Exception Handling:**
  - try...catch, finally, throw

## 21. How do break and continue statements work in loops?

### **Answer:**

- **break:** Terminates the loop entirely.

```
for (let i = 0; i < 5; i++) {  
  if (i === 3) break;  
  console.log(i);  
}  
// Output: 0, 1, 2
```

- **continue:** Skips the current iteration and moves to the next one.

```
for (let i = 0; i < 5; i++) {  
  if (i === 3) continue;  
  console.log(i);  
}  
// Output: 0, 1, 2, 4
```

## **22. How does JavaScript's event loop affect control flow?**

**Answer:** The **event loop** ensures that JavaScript executes code in a non-blocking way. Tasks are executed in the following order:

1. **Call Stack:** Executes synchronous code.
2. **Task Queue:** Executes asynchronous callbacks after the stack is cleared.
3. **Microtasks:** Promises and async/await are executed before the task queue.

```
console.log("Start");  
  
setTimeout(() => console.log("Timeout"), 0);  
  
Promise.resolve().then(() => console.log("Promise"));  
  
console.log("End");  
  
// Output:  
// Start  
// End  
// Promise  
// Timeout
```

### **Array:**

## **23. What are some common array methods in JavaScript?**

### **Answer:**

<b>Method</b>	<b>Description</b>	<b>Example</b>
push()	Adds elements to the end of the array	arr.push(4)

Method	Description	Example
pop()	Removes the last element	arr.pop()
shift()	Removes the first element	arr.shift()
unshift()	Adds elements to the beginning	arr.unshift(0)
splice()	Adds/removes elements at a specific index	arr.splice(1, 2)
slice()	Returns a subarray without modifying the original	arr.slice(1, 3)
map()	Creates a new array by applying a function	arr.map(x => x * 2)
filter()	Filters elements based on a condition	arr.filter(x => x > 2)
reduce()	Reduces the array to a single value	arr.reduce((a, b) => a + b, 0)
forEach()	Iterates through the array	arr.forEach(x => console.log(x))
includes()	Checks if the array contains a value	arr.includes(3)
find()	Returns the first element that matches a condition	arr.find(x => x > 2)

## 24. What is the difference between splice() and slice()?

Answer:

Feature	splice()	slice()
<b>Modification</b>	Modifies the original array	Does not modify the original array
<b>Purpose</b>	Adds/removes elements at a specific index	Extracts a portion of the array
<b>Example</b>	arr.splice(1, 2)	arr.slice(1, 3)

## 25. What is the difference between for...of and for...in with arrays?

Answer:

Feature	for...of	for...in
<b>Iterates Over</b>	Values of the array	Indices of the array
<b>Example</b>	for (let value of arr)	for (let index in arr)

Example:

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

for (let value of arr) {
  console.log(value); // Output: 10, 20, 30
}

for (let index in arr) {
  console.log(index); // Output: 0, 1, 2
}
```

## 26. What is the purpose of the map() method in JavaScript?

**Answer:**

1. The map() method creates a new array by applying a function to every element of the original array.
2. It does not modify the original array but returns a transformed version of it.
3. Useful for performing operations like calculations, formatting, or applying transformations on arrays.
4. Works on all elements of the array, and the original array remains unchanged.

```
const numbers = [1, 2, 3];
const squares = numbers.map(num => num * num);
console.log(squares); // Output: [1, 4, 9]
```

## 27. How does the find() method work, and how is it different from filter()?

**Answer:**

1. **find() method:**
  - o Returns the first element that satisfies a specified condition.
  - o Stops searching as soon as it finds a match.
  - o Returns the value directly, not in an array.
2. **filter() method:**
  - o Returns all elements that satisfy a condition as a new array.
  - o Does not stop after finding one match.
3. Use find() when you only need the first match.
4. Use filter() when you need all matches.

```
const numbers = [1, 2, 3, 4];
const firstEven = numbers.find(num => num % 2 === 0); // Finds the first even number
console.log(firstEven); // Output: 2

const allEvens = numbers.filter(num => num % 2 === 0); // Finds all even numbers
console.log(allEvens); // Output: [2, 4]
```

## 28. What is the purpose of the flat() method?

**Answer:**

1. The flat() method is used to flatten nested arrays into a single-level array.
2. By default, it flattens one level of nesting.
3. You can specify the depth of flattening as an argument.
4. To fully flatten a deeply nested array, use flat(Infinity).

**Example:**

```
const arr = [1, [2, [3, [4]]]];
```

```
console.log(arr.flat(2)); // Output: [1, 2, 3, [4]]
```

## 29. How does some() differ from every()?

**Answer:**

1. **some() method:**
  - o Checks if **at least one** element satisfies the condition.
  - o Stops checking once a match is found.
  - o Returns true or false.
2. **every() method:**
  - o Checks if **all elements** satisfy the condition.
  - o Stops checking once a mismatch is found.
  - o Returns true or false.
3. Both methods are short-circuiting, meaning they stop as soon as the result is determined.
4. some() is useful for partial matches, while every() is for all matches.

**Example:**

```
const numbers = [1, 2, 3, 4];
console.log(numbers.some(num => num > 3)); // Output: true (at least one element > 3)
console.log(numbers.every(num => num > 3)); // Output: false (not all elements > 3)
```

## 30. How can you check if a value exists in an array?

**Answer:**

1. Use the includes() method to check if an array contains a specific value.
2. It returns true if the value is found and false otherwise.
3. Works for both primitive values and exact matches.
4. Does not work for object references unless the reference itself matches.

**Example:**

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

## 31. What is the difference between forEach() and map()?

**Answer:**

The forEach() and map() methods are both used to iterate over arrays, but they serve different purposes and have distinct characteristics:

1. **Purpose:**
  - o forEach() is used for **executing a function** on each element of the array. It is commonly used for side effects, such as logging or modifying external variables, but it does not return a new array.

- map() is used for **transforming data**. It applies a function to each element of the array and returns a new array with the transformed values.
- 2. Return Value:**
- forEach() does not return anything (its return value is undefined). It is designed for actions, not for creating a new array.
  - map() always returns a new array with the same length as the original, containing the results of the callback function.
- 3. Chaining:**
- forEach() cannot be chained because it does not return a new array.
  - map() can be chained with other array methods like filter() or reduce() because it returns a new array.

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

#### **4. Performance:**

- In most cases, there is no significant performance difference, but map() is better suited for creating transformed arrays since it is more declarative.

### **Function:**

## **32. What is the difference between function declarations and function expressions?**

### **Answer:**

#### **Function Declaration:**

1. A named function is defined using the function keyword.
2. It is hoisted, meaning it can be used before its declaration in the code.

#### **Example:**

```
console.log(square(4)); // Output: 16
function square(num) {
  return num * num;
}
```

#### **Function Expression:**

1. A function is assigned to a variable, often anonymous.
2. It is not hoisted, meaning it cannot be used before its definition in the code.

#### **Example:**

```
const square = function(num) {
  return num * num;
};
```

```
console.log(square(4)); // Output: 16
```

### Key Differences:

4. Function declarations are defined at parse time (hoisting), while function expressions are defined at runtime.
5. Function declarations must have a name, whereas function expressions can be anonymous.

## 33. What are arrow functions, and how do they differ from regular functions?

Answer:

### Arrow Functions:

1. Introduced in ES6 as a concise syntax for defining functions.
2. They use the `=>` (arrow) syntax instead of the `function` keyword.
3. Do not have their own `this` context; they inherit this from their surrounding scope.
4. Do not have an `arguments` object, unlike regular functions.

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5
```

### Differences from Regular Functions:

1. Arrow functions cannot be used as constructors (do not work with the `new` keyword).
2. They do not bind their own `this` context, making them suitable for use in callbacks.
3. They provide a more concise syntax, especially for single-expression functions.

Example:

```
function regularFunction() {
  console.log(this); // Refers to the function's context
}
const arrowFunction = () => {
  console.log(this); // Inherits from the surrounding scope
};
regularFunction.call({ key: "value" });
arrowFunction.call({ key: "value" });
```

## 34. What is the difference between `call()`, `apply()`, and `bind()` in JavaScript?

Answer:

These methods are used to set the `this` context explicitly in JavaScript:

1. `call()`: Invokes a function immediately with the provided `this` and arguments passed individually.

Example:

```
function greet(greeting) {
```

```
    console.log(` ${greeting}, ${this.name}`);
}
const person = { name: "Mahesh" };
greet.call(person, "Hello"); // Output: Hello, Mahesh
```

**2. apply():** Similar to call(), but arguments are passed as an array.

**Example:**

```
greet.apply(person, ["Hi"]); // Output: Hi, Mahesh
```

**3. bind():** Does not invoke the function immediately. Instead, it creates a new function with the specified this.

```
const boundGreet = greet.bind(person, "Hey");
boundGreet(); // Output: Hey, Mahesh
```

### 35. What is the difference between return and console.log() in a function?

**Answer:**

**return:**

1. Ends the function's execution and sends the result back to the caller.
2. The returned value can be stored in a variable for later use.
3. Stops further code execution within the function.
4. Used to pass data back to the point where the function was called.
5. Only one value can be returned per function call.

```
function add(a, b) {
  console.log(a + b); // Logs the result
  return a + b;      // Returns the result
}
```

**console.log():**

1. Outputs a message or value to the console for debugging purposes.
2. Does not stop the function's execution.
3. Does not return any value.
4. Primarily used to track or debug code behavior.
5. Cannot be used for further calculations or assignments.

```
const result = add(2, 3); // Logs 5 and returns 5
console.log(result); // Output: 5
```

### 36. How do you handle default parameters in JavaScript functions?

**Answer:**

1. Default parameters allow a function to use predefined values when no argument or undefined is provided for the parameter.
2. Helps avoid errors caused by missing arguments.
3. Default values are assigned in the function definition using the = operator.
4. Introduced in ES6 for better handling of optional parameters.
5. Makes code cleaner and more readable by eliminating the need for conditional checks for undefined parameters.

```
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}

console.log(greet()); // Output: Hello, Guest!
console.log(greet("Mahesh")); // Output: Hello, Mahesh!
```

### Object:

#### **37. What is the difference between dot notation and bracket notation in accessing object properties?**

##### **Answer:**

Dot notation (.) and bracket notation ([]) are two ways to access properties in an object:

1. **Dot Notation:** Used when the property name is a valid identifier (e.g., no spaces or special characters).

```
const person = { name: "Mahesh" };
console.log(person.name); // Output: Mahesh
```

2. **Bracket Notation:** Used when the property name is dynamic, stored in a variable, or contains special characters.

```
const person = { "full name": "Mahesh" };
console.log(person["full name"]); // Output: Mahesh
```

Bracket notation provides more flexibility but is slightly less readable compared to dot notation.

#### **38. What is the difference between Object.keys(), Object.values(), and Object.entries()?**

##### **Answer:**

These methods provide different ways to access an object's data:

**Object.keys():** Returns an array of the object's property names (keys).

```
const obj = { a: 1, b: 2 };
```

```
console.log(Object.keys(obj)); // Output: ['a', 'b']
```

**Object.values()**: Returns an array of the object's property values.

```
console.log(Object.values(obj)); // Output: [1, 2]
```

**Object.entries()**: Returns an array of key-value pairs as subarrays.

```
console.log(Object.entries(obj)); // Output: [['a', 1], ['b', 2]]
```

These methods are helpful for iterating over objects or transforming them into other structures.

### 39. What is the purpose of Object.freeze() and Object.seal()?

**Answer:**

- \*\*

**Object.freeze()\*\*:** Prevents modifications to an object. You cannot add, delete, or change its properties. The object becomes immutable.

**Example:**

```
const obj = { a: 1 };
Object.freeze(obj);
obj.a = 2; // This has no effect
obj.b = 3; // This also has no effect
console.log(obj); // Output: { a: 1 }
```

**Object.seal()**: Prevents adding or deleting properties but allows modifying existing ones.

**Example:**

```
const obj = { a: 1 };
Object.seal(obj);
obj.a = 2; // Allowed
obj.b = 3; // Not allowed
console.log(obj); // Output: { a: 2 }
```

Both methods are used to control the mutability of objects for maintaining strict data integrity.

### 40. What is the difference between Object.create() and the new keyword?

**Answer:**

**Object.create()**: Creates a new object with a specified prototype. It gives you fine-grained control over inheritance.

**Example:**

```
const proto = { greet: () => "Hello" };
const obj = Object.create(proto);
console.log(obj.greet()); // Output: Hello
```

**new Keyword:** Creates an object instance from a constructor function. It initializes the object using the constructor logic.

**Example:**

```
function Person(name) {
  this.name = name;
}
const person = new Person("Mahesh");
console.log(person.name); // Output: Mahesh
```

## DOM:

### 41. What is the difference between innerHTML, innerText, and textContent?

**Answer:**

**1. innerHTML:** Gets or sets the HTML content inside an element, including HTML tags. It is useful for rendering complex HTML structures but can expose your code to security risks like XSS if used with untrusted content.

```
element.innerHTML = "<strong>Bold Text</strong>";
```

**2. innerText:** Gets or sets the text inside an element, excluding hidden elements. It reflects what is rendered visually on the page.

```
element.innerText = "Visible Text";
```

**3. textContent:** Gets or sets the text inside an element, including hidden elements, but ignores any HTML tags.

```
element.textContent = "Plain Text";
```

Choose the appropriate property based on whether you need plain text, formatted text, or raw HTML.

### 42. How do you add, remove, or modify DOM elements dynamically?

**Answer:**

You can manipulate DOM elements dynamically using various methods:

**Create a new element:**

```
const newElement = document.createElement("div");
```

```
newElement.textContent = "Hello!";
```

**Add the element to the DOM:**

```
document.body.appendChild(newElement);
```

**Remove an element:**

```
const element = document.getElementById("removeMe");
element.remove();
```

**Modify attributes or styles:**

```
newElement.setAttribute("id", "greeting");
newElement.style.color = "blue";
```

These methods allow you to create interactive and dynamic web pages.

### 43. How do you manipulate classes on DOM elements?

**Answer:**

1. Use the classList property to manage CSS classes on an element.
2. The classList API provides methods to add, remove, toggle, or check for a class.
3. It simplifies class manipulations compared to working directly with the className property.
4. Allows easy addition of multiple classes in one statement.
5. Provides a cleaner and more efficient way to manage classes dynamically.

```
const element = document.getElementById("myElement");

// Add a class
element.classList.add("active");

// Remove a class
element.classList.remove("inactive");

// Toggle a class
element.classList.toggle("hidden");

// Check if a class exists
console.log(element.classList.contains("active")); // Output: true
```

The classList API simplifies class manipulations compared to working directly with the className property

**Event:**

#### 44. What is event delegation, and why is it useful in the DOM?

**Answer:**

1. Event delegation is a technique where a single event listener is attached to a parent element to handle events triggered by its child elements.
2. It leverages the concept of event bubbling, where events propagate from child elements to their parents.
3. Reduces the number of event listeners needed, improving performance, especially for dynamically created elements.
4. Simplifies event handling for dynamically added or removed child elements.
5. Uses the event.target property to identify the specific child element that triggered the event.

```
document.getElementById("parent").addEventListener("click", function (event) {  
  if (event.target && event.target.tagName === "BUTTON") {  
    console.log(`Button clicked: ${event.target.textContent}`);  
  }  
});
```

#### 45. How do you handle events in the DOM?

**Answer:**

Events in the DOM can be handled using:

1. **Inline Handlers:** Specified directly in HTML (not recommended for large applications).

```
<button onclick="alert('Clicked!')">Click Me</button>;
```

**Directly Setting Event Handlers:** Assign a function to an element's event property.

```
const button = document.getElementById("btn");  
button.onclick = () => alert("Clicked!");
```

**Using addEventListener():** Preferred for modern JavaScript because it allows multiple handlers for the same event.

```
button.addEventListener("click", () => alert("Clicked!"));
```

The addEventListener() method also supports options like once (for one-time events) and capture (for capturing phase).

#### 46. What is the difference between capturing and bubbling in event propagation?

**Answer:**

Event propagation describes how events flow through the DOM tree. It occurs in three phases:

- Capturing Phase:** The event starts at the root and travels down to the target element.
- Target Phase:** The event reaches the target element.
- Bubbling Phase:** The event travels back up from the target to the root.

By default, events are handled in the bubbling phase unless the capture option is set to true in addEventListener().

```
document
  .getElementById("parent")
  .addEventListener("click", () => console.log("Parent clicked"), true); // Capturing
document
  .getElementById("child")
  .addEventListener("click", () => console.log("Child clicked")); // Bubbling
```

## 46. How do you prevent the default behaviour of an event?

**Answer:**

- Use the preventDefault() method to stop the browser's default action for an event.
- It is commonly used to prevent actions like link navigation, form submission, or context menu display.
- The method is invoked on the event object within the event handler.
- It allows developers to implement custom behavior in place of the default.
- Does not stop event propagation; it only blocks the default action associated with the event.

**Example:**

```
document.querySelector("a").addEventListener("click", function (event) {
  event.preventDefault();
  console.log("Default navigation prevented!");
});
```

[var, let, and const](#):

## 47. What is the difference between var, let, and const in JavaScript?

**Answer:**

- Scope:**
  - var is function-scoped. It is visible within the function where it is declared and ignored in block scopes.
  - let and const are block-scoped, meaning they are limited to the block, statement, or expression in which they are declared.
- Re-declaration:**
  - var allows re-declaration within the same scope.
  - let does not allow re-declaration in the same scope.
  - const also does not allow re-declaration.
- Initialization:**

- Variables declared with var are initialized as undefined if not explicitly assigned.
- let and const do not initialize the variable until their declaration is executed (temporal dead zone).

#### 4. Mutability:

- Variables declared with var and let can be reassigned.
- const variables cannot be reassigned, though their properties can be modified if they reference an object.

```
function testScope() {
  if (true) {
    var x = 10;
    let y = 20;
    const z = 30;
  }
  console.log(x); // Output: 10 (function-scoped)
  // console.log(y); // Error: y is not defined (block-scoped)
  // console.log(z); // Error: z is not defined (block-scoped)
}
testScope();
```

### 48. What is the Temporal Dead Zone (TDZ) in JavaScript?

#### Answer:

1. The Temporal Dead Zone (TDZ) is the time between the start of a block and when a variable declared with let or const is initialized.
2. During this period, accessing the variable results in a ReferenceError.
3. Unlike var, variables declared with let or const are not initialized as undefined at the start of execution.
4. The TDZ applies only to variables declared with let or const, not var.
5. Ensures safer coding practices by preventing accidental use of variables before their declaration.

```
console.log(x); // Output: undefined (var is hoisted)
console.log(y); // Error: Cannot access 'y' before initialization
var x = 10;
let y = 20;
```

### 49. Why should var be avoided in modern JavaScript?

#### Answer:

1. **Function Scope:** var is function-scoped, which can lead to unintended behavior when used inside block statements.
2. **Hoisting Issues:** Variables declared with var are hoisted and initialized as undefined, leading to potential bugs.
3. **Re-declaration:** var allows re-declaration, which can overwrite existing variables and cause confusion.

```

if (true) {
  var x = 10;
}
console.log(x); // Output: 10 (unexpected exposure outside block)

if (true) {
  let y = 20;
}
// console.log(y); // Error: y is not defined (block-scoped)

```

## 50. How does hoisting differ for var, let, and const?

**Answer:**

1. **var:** Variables declared with var are hoisted to the top of their scope and initialized as undefined. This can lead to unexpected results.
2. **let and const:** These are hoisted but remain uninitialized until their declaration is executed (TDZ).

```

console.log(a); // Output: undefined (var is hoisted)
var a = 10;

console.log(b); // Error: Cannot access 'b' before initialization
let b = 20;

console.log(c); // Error: Cannot access 'c' before initialization
const c = 30;

```

Understanding hoisting helps avoid runtime errors and unexpected behavior.

## 51. How do block scope and function scope affect variable visibility?

**Answer:**

- **Block Scope (let and const):** Variables declared with let or const are limited to the block in which they are defined. They are not accessible outside the block.
- **Function Scope (var):** Variables declared with var are accessible throughout the function in which they are defined, even if declared inside a block.

```

if (true) {
  var a = 10;
  let b = 20;
  const c = 30;
}
console.log(a); // Output: 10 (function-scoped)
// console.log(b); // Error: b is not defined (block-scoped)
// console.log(c); // Error: c is not defined (block-scoped)

```

Using block-scoped variables (let and const) helps avoid polluting the outer scope.

### Spread and Rest:

## 52. What is the spread operator in JavaScript, and how is it used?

### **Answer:**

The spread operator (...) is a syntax introduced in ES6 that allows you to spread the elements of an array, object, or iterable into individual elements. It is commonly used for cloning, merging, or passing arguments to functions.

### **Examples:**

#### **In Arrays:**

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

#### **In Objects:**

```
const obj = { a: 1, b: 2 };
const newObj = { ...obj, c: 3 };
console.log(newObj); // Output: { a: 1, b: 2, c: 3 }
```

#### **In Functions:**

```
const numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // Output: 3
```

## 53. How does the spread operator differ from the rest operator?

### **Answer:**

The spread operator (...) is used to **unpack elements** from arrays or objects, while the rest operator (...) is used to **pack elements** into an array or object.

### **Example of Spread:**

```
const arr = [1, 2, 3];
const newArr = [...arr, 4, 5]; // Spreads elements into a new array
console.log(newArr); // Output: [1, 2, 3, 4, 5]
```

### **Example of Rest:**

```
function sum(...numbers) {
  // Packs arguments into an array
  return numbers.reduce((acc, curr) => acc + curr, 0);
}
```

```
console.log(sum(1, 2, 3)); // Output: 6
```

The spread operator extracts elements, whereas the rest operator gathers them.

## 54. How do you clone an array or object using the spread operator?

### Answer:

The spread operator can be used to create a shallow copy of arrays or objects. This means the copied array or object shares references with the original for nested structures.

### Cloning an Array:

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

### Cloning an Object:

```
const obj = { a: 1, b: 2 };
const clonedObj = { ...obj };
clonedObj.c = 3;
console.log(obj); // Output: { a: 1, b: 2 }
console.log(clonedObj); // Output: { a: 1, b: 2, c: 3 }
```

Use this approach for shallow copies, but for deeply nested objects, a deep cloning method is required.

## 55. What are the limitations of the spread operator?

### Answer:

**Shallow Copy Only:** The spread operator creates shallow copies of objects and arrays. Nested structures will still share references with the original.

```
const obj = { a: { b: 1 } };
const cloned = { ...obj };
cloned.a.b = 2;
console.log(obj.a.b); // Output: 2 (shared reference)
```

**Not Applicable to Non-Iterables:** The spread operator only works with iterables (e.g., arrays, strings, sets). Applying it to non-iterable objects throws an error.

```
const obj = 123;
// const result = [...obj]; // Error: obj is not iterable
```

To handle deep copies, use libraries like lodash or structuredClone.

## 56. How can you merge arrays or objects using the spread operator?

**Answer:**

The spread operator provides a simple way to merge arrays and objects by spreading their elements or properties into a new structure.

**Merging Arrays:**

```
const arr1 = [1, 2];
const arr2 = [3, 4];
const mergedArr = [...arr1, ...arr2];
console.log(mergedArr); // Output: [1, 2, 3, 4]
```

**Merging Objects:**

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj); // Output: { a: 1, b: 3, c: 4 }
```

In objects, later properties overwrite earlier ones if there are conflicts.

**Destructuring:**

## 57. What is the difference between array destructuring and object destructuring?

**Answer:**

**Array Destructuring:** The order of elements matters, as values are assigned based on their positions in the array.

```
const arr = [1, 2, 3];
const [a, b] = arr;
console.log(a, b); // Output: 1, 2
```

**Object Destructuring:** The order does not matter, as values are matched by property names.

```
const obj = { name: "Mahesh", age: 25 };
const { age, name } = obj;
console.log(name, age); // Output: Mahesh, 25
```

Array destructuring is positional, while object destructuring is based on property names.

## 58. How do you use default values with destructuring?

### Answer:

You can assign default values to variables during destructuring to handle undefined or missing values. If the property or element is not found, the default value is used instead.

### Example:

```
// Array Destructuring with Defaults
const arr = [1];
const [a, b = 2] = arr;
console.log(a, b); // Output: 1, 2

// Object Destructuring with Defaults
const obj = { name: "Mahesh" };
const { name, age = 25 } = obj;
console.log(name, age); // Output: Mahesh, 25
```

Default values ensure robustness when working with incomplete data structures.

## 59. How do you combine rest and destructuring?

### Answer:

You can use the rest operator (...) with destructuring to collect the remaining properties or elements into a separate variable.

### Example with Objects:

```
const obj = { a: 1, b: 2, c: 3 };
const { a, ...rest } = obj;
console.log(a); // Output: 1
console.log(rest); // Output: { b: 2, c: 3 }
```

Example with Arrays:

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

The rest operator gathers remaining values into a new object or array.

## 60. How can destructuring be used with default function arguments?

**Answer:**

You can provide default values for function arguments during destructuring. This is useful for handling optional arguments in functions.

**Example:**

```
function greet({ name = "Guest", age = 0 } = {}) {  
  return `Hello, ${name}. You are ${age} years old.`;  
}  
console.log(greet({ name: "Mahesh" })); // Output: Hello, Mahesh. You are 0 years old.  
console.log(greet()); // Output: Hello, Guest. You are 0 years old.
```

Providing defaults ensures functions behave correctly even with missing or incomplete arguments.

**Prototype:****61. What is the difference between `__proto__` and `prototype`?****Answer:****1. `__proto__`:**

- A property of an object that points to its prototype.
- Used for accessing the prototype chain dynamically.

**2. `prototype`:**

- A property of constructor functions that is used to define properties and methods for instances created by the constructor.
- It's not directly accessible on instances but determines the `__proto__` of the instances.

**Example:**

```
function Person() {}  
const mahesh = new Person();  
  
console.log(Person.prototype); // Output: Prototype object of Person  
console.log(mahesh.__proto__); // Output: Same as Person.prototype
```

The `__proto__` links the instance to its constructor's prototype.

**62. How can you create objects without a prototype?****Answer:**

You can create objects without a prototype using `Object.create(null)`. This is useful when you need a truly plain object without inherited properties.

```
const obj = Object.create(null);  
console.log(obj); // Output: {}
```

```
console.log(obj.toString()); // Output: undefined
```

Here, obj has no prototype, so it doesn't inherit methods like `toString` or `hasOwnProperty`.

### 63. What is the difference between prototypal inheritance and classical inheritance?

**Answer:**

- **Prototypal Inheritance:** Objects inherit directly from other objects using prototypes. It is more flexible and simpler compared to classical inheritance.
- **Classical Inheritance:** Objects are instantiated from classes (constructors in JavaScript). This is more rigid and mimics inheritance in languages like Java or C++.

Example of Prototypal Inheritance:

```
const animal = { eat: () => "Eating" };
const dog = Object.create(animal);
console.log(dog.eat()); // Output: Eating
```

Example of Classical Inheritance:

```
class Animal {
  eat() {
    return "Eating";
  }
}
class Dog extends Animal {}
const dog = new Dog();
console.log(dog.eat()); // Output: Eating
```

Classes:

### 64. What is the difference between a class and a constructor function?

**Answer:**

1. **Syntax:** Classes are more concise and resemble traditional OOP languages. Constructor functions are less structured and use prototypes for inheritance.
2. **Hoisting:** Classes are not hoisted, meaning you cannot use a class before declaring it. Constructor functions are hoisted, but only their declaration.
3. **strict mode:** Classes always operate in strict mode, which helps prevent certain common bugs.
4. **Static Methods:** Classes have built-in support for static methods, while constructor functions require manual implementation.

Example of Constructor Function:

```
function Person(name) {
  this.name = name;
```

```
}
```

```
Person.prototype.greet = function () {
  return `Hello, ${this.name}`;
};

const mahesh = new Person("Mahesh");
console.log(mahesh.greet()); // Output: Hello, Mahesh
```

## 65. What is the purpose of the super keyword in classes?

### Answer:

The super keyword is used to call the constructor or methods of a parent class. It is essential when a subclass needs to initialize properties defined in the parent class or when it wants to extend a method from the parent class.

### Example:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Calls the parent class constructor
    this.breed = breed;
  }
  getDetails() {
    return `${this.name} is a ${this.breed}`;
  }
}

const dog = new Dog("Buddy", "Labrador");
console.log(dog.getDetails()); // Output: Buddy is a Labrador
```

## 66. What is the difference between class and function constructors in terms of inheritance?

### Answer:

- Syntax:** Classes use extends for inheritance, while functions rely on manually setting the prototype using Object.create() or Object.setPrototypeOf().
- super Keyword:** Classes use super for parent class constructor calls. Functions need explicit calls to the parent constructor.
- Code Readability:** Classes offer a cleaner and more intuitive syntax compared to prototype-based inheritance in functions.

### **Example with Class:**

```
class Animal {  
    constructor(name) {  
        this.name = name;  
    }  
}  
class Dog extends Animal {  
    constructor(name) {  
        super(name);  
    }  
}
```

### **Example with Function:**

```
function Animal(name) {  
    this.name = name;  
}  
function Dog(name) {  
    Animal.call(this, name);  
}  
Dog.prototype = Object.create(Animal.prototype);
```

### **Arrow Function:**

#### **67. How do arrow functions differ from traditional functions in terms of this?**

#### **Answer:**

1. **Arrow functions** do not have their own this context; they inherit this from the surrounding lexical scope.
2. **Traditional functions** have their own this, determined by how the function is called (runtime binding).
3. Arrow functions are particularly useful in callbacks or event listeners, where this might otherwise be lost.
4. Arrow functions do not bind their own this when used in methods or inside constructors.
5. Traditional functions are suitable when dynamic this binding is required.

#### **Example:**

```
function Traditional() {  
    this.value = 10;  
    setTimeout(function () {  
        console.log(this.value); // Output: undefined (or throws an error in strict mode)  
    }, 100);  
}  
  
function ArrowFunction() {
```

```
this.value = 10;
setTimeout(() => {
  console.log(this.value); // Output: 10 (inherits `this` from ArrowFunction)
}, 100);
}

new Traditional();
new ArrowFunction();
```

## 68. Can arrow functions be used as constructors?

**Answer:**

1. No, arrow functions cannot be used as constructors.
2. They lack their own this context, which is essential for creating objects with the new keyword.
3. Arrow functions do not have a prototype property, making object instantiation impossible.
4. Attempting to use the new keyword with an arrow function will throw an error.
5. Regular functions or classes must be used when a constructor is required.

**Example:**

```
const ArrowConstructor = () => {};
// const obj = new ArrowConstructor(); // Error: ArrowConstructor is not a constructor
```

If a constructor is required, you must use a regular function or class.

## 69. Can arrow functions have default parameters?

**Answer:**

1. Yes, arrow functions support default parameters.
2. Default parameters allow you to assign a value to a parameter when no argument or undefined is provided.
3. The syntax for default parameters is the same as for traditional functions.
4. Helps make arrow functions cleaner and reduces the need for additional checks or conditions.
5. Default parameters can be combined with rest parameters or destructuring for enhanced functionality.

**Example:**

```
const greet = (name = "Guest") => `Hello, ${name}!`;
console.log(greet()); // Output: Hello, Guest!
console.log(greet("Mahesh")); // Output: Hello, Mahesh!
```

## 70. What are the limitations of arrow functions?

### **Answer:**

1. **No this Context:** Arrow functions inherit this from the surrounding lexical scope, making them unsuitable for dynamic this contexts like object methods.
2. **No arguments Object:** Arrow functions do not have their own arguments. Use the rest operator (...args) instead.
3. **Cannot Be Used as Constructors:** Arrow functions cannot be used with the new keyword to create objects.
4. **No super Support:** Arrow functions cannot use super directly, which is crucial for extending classes.

### Example of Limitation:

```
const obj = {
  value: 10,
  method: () => {
    console.log(this.value); // Output: undefined (not bound to `obj`)
  },
};
obj.method();
```

### Closures:

#### **71. What are the practical uses of closures?**

### **Answer:**

1. **Data Encapsulation:** Closures help create private variables, mimicking the behavior of private properties.

```
function Counter() {
  let count = 0;
  return {
    increment() {
      count++;
    },
    getCount() {
      return count;
    },
  };
}
const counter = Counter();
counter.increment();
console.log(counter.getCount()); // Output: 1
```

**Function Factories:** Closures can create functions with preset configurations.

```
function multiplier(factor) {
```

```
return function (number) {  
    return number * factor;  
};  
  
const double = multiplier(2);  
console.log(double(5)); // Output: 10
```

**Event Handlers:** Closures allow event handlers to retain state from their creation context.

## 72. How do closures work with asynchronous code?

**Answer:**

Closures retain their reference to variables even when the containing function has completed execution. This is useful in asynchronous operations like setTimeout or promises.

**Example:**

```
function delayedMessage(message, delay) {  
    setTimeout(() => {  
        console.log(message); // Accesses `message` from the outer scope  
    }, delay);  
}  
delayedMessage("Hello after 1 second", 1000); // Output: Hello after 1 second
```

The closure ensures the message variable remains accessible to the callback even after delayedMessage has finished executing.

## 73. How are callbacks used in asynchronous programming?

**Answer:**

Callbacks are crucial in asynchronous programming, as they execute once the asynchronous operation completes, ensuring non-blocking code execution.

**Example:**

```
setTimeout(() => {  
    console.log("Executed after 2 seconds");  
}, 2000);  
console.log("This runs first");
```

Output:

```
This runs first  
Executed after 2 seconds
```

Here, the callback passed to setTimeout runs after the specified delay, while the main thread continues.

## 74. What are common issues with callbacks, and how can they be solved?

**Answer:**

1. **Callback Hell:** When multiple nested callbacks create deeply indented code, making it hard to read and maintain.

```
fetchData((data) => {
  processData(data, (result) => {
    saveData(result, (status) => {
      console.log("Data saved!");
    });
  });
});
```

**Solution:** Use Promises or async/await to manage asynchronous code.

**Error Handling:** Errors in callbacks can go unnoticed if not explicitly handled.

```
function asyncOperation(callback) {
  try {
    // Simulating an error
    throw new Error("Something went wrong");
  } catch (error) {
    callback(error);
  }
}
asyncOperation((err) => {
  if (err) console.error(err.message); // Output: Something went wrong
});
```

## 75. What is the difference between callbacks and Promises?

**Answer:**

1. **Readability:** Promises simplify chaining and improve code readability compared to nested callbacks.
2. **Error Handling:** Promises handle errors using .catch() or try-catch in async/await.
3. **Multiple Handlers:** Promises can have multiple .then() handlers, while callbacks are executed once.

**Example with Promises:**

```
fetchData()
  .then(processData)
  .then(saveData)
  .then(() => console.log("Data saved!"))
  .catch((error) => console.error(error));
```

Promises reduce the complexity of managing asynchronous operations compared to callbacks.

### **HOF => Map, Filter, Reduce:**

## **76. What are the advantages of using Higher-Order Functions?**

### **Answer:**

1. **Code Reusability:** HOFs allow you to define logic in one place and reuse it with different callbacks.
2. **Abstraction:** They help abstract repetitive tasks, like iteration or filtering, into reusable functions.
3. **Improved Readability:** They simplify code by removing boilerplate, especially when working with arrays or asynchronous tasks.

### **Example of Abstraction:**

```
function processArray(arr, callback) {  
  return arr.map(callback);  
}  
  
const numbers = [1, 2, 3];  
console.log(processArray(numbers, num => num * 2)); // Output: [2, 4, 6]
```

## **77. How do Higher-Order Functions work with Promises?**

### **Answer:**

HOFs like `.then()` and `.catch()` in Promises take callback functions as arguments to handle asynchronous tasks. This allows chaining operations and handling errors cleanly.

### **Example:**

```
fetch("https://api.example.com/data")  
  .then((response) => response.json())  
  .then((data) => console.log(data))  
  .catch((error) => console.error("Error:", error));
```

Here, `.then()` and `.catch()` are Higher-Order Functions that accept callbacks for success and error handling.

## **78. How does map() handle empty slots in arrays?**

### **Answer:**

If an array contains empty slots (e.g., `[1, , 3]`), the `map()` method skips these slots but maintains their positions in the new array.

### **Example:**

```
const arr = [1, , 3];
const result = arr.map((x) => x || 0); // Replace empty slots with 0
console.log(result); // Output: [1, 0, 3]
```

Empty slots are ignored by map() unless explicitly handled in the callback.

## 79. What happens if filter() does not find any matching elements?

**Answer:**

If no elements match the condition, the filter() method returns an empty array.

**Example:**

```
const numbers = [1, 2, 3];
const greaterThanTen = numbers.filter((num) => num > 10);
console.log(greaterThanTen); // Output: []
```

## 80. How do the initialValue and accumulator work in reduce()?

**Answer:**

- **initialValue:** The starting value for the accumulator. If not provided, the first element of the array is used as the initial value.
- **accumulator:** Holds the cumulative result of the callback function across iterations.

**Example:**

```
const numbers = [1, 2, 3, 4];

// Without initialValue
const sum1 = numbers.reduce((acc, curr) => acc + curr);
console.log(sum1); // Output: 10

// With initialValue
const sum2 = numbers.reduce((acc, curr) => acc + curr, 5);
console.log(sum2); // Output: 15
```

## 81. How is the map() function an example of a Higher-Order Function?

**Answer:**

The map() function is a built-in HOF that applies a callback function to every element of an array and returns a new array with the results. It demonstrates how HOFs abstract behavior by allowing custom logic to be passed as a function.

**Example:**

```
const numbers = [1, 2, 3, 4];
```

```
const squared = numbers.map((num) => num * num);
console.log(squared); // Output: [1, 4, 9, 16]
```

Here, the map() function takes the callback (num => num \* num) as an argument to apply the squaring logic to each element.

## 82. How does filter() work as a Higher-Order Function?

### Answer:

The filter() function takes a callback that returns a boolean value. It creates a new array containing only the elements that pass the test defined in the callback.

### Example:

```
const numbers = [1, 2, 3, 4, 5];
const evens = numbers.filter((num) => num % 2 === 0);
console.log(evens); // Output: [2, 4]
```

In this example, filter() uses the callback (num => num % 2 === 0) to decide which numbers to include in the result.

## 83. How is reduce() an example of a Higher-Order Function?

### Answer:

The reduce() method takes a callback function and an optional initial value. It applies the callback to each element of the array, accumulating a single output value.

### Example:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // Output: 10
```

Here, reduce() is a HOF because it takes a callback (acc, curr) => acc + curr to define the accumulation logic.

### Debouncing or Throttling:

## 84. What is debouncing in JavaScript?

### Answer:

1. **Debouncing** is a technique to limit the number of times a function is executed by ensuring it is only invoked after a specified delay.

2. It is used to improve performance by reducing the frequency of event handling in scenarios like window resizing, button clicks, or keystrokes.
3. Prevents a function from being called repeatedly in rapid succession, such as during continuous user interactions.
4. Involves resetting a timer (setTimeout) every time the event is triggered, delaying the function execution until no events occur during the delay period.
5. Commonly implemented in scenarios where repeated executions of a function can degrade performance, such as handling scroll or resize events.

**Example:**

```
function debounce(func, delay) {
  let timeout;
  return function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), delay);
  };
}

// Example Usage
const log = () => console.log("Debounced function executed");
const debouncedLog = debounce(log, 1000);

window.addEventListener("resize", debouncedLog);
```

Here, the log function is called only after the user stops resizing the window for 1 second.

## 85. What are the practical use cases of debouncing?

**Answer:**

1. **Search Bar Suggestions:** Sending API requests only after the user has stopped typing.
2. **Window Resize Events:** Avoiding performance-intensive operations when the window is continuously resized.
3. **Form Validations:** Validating input fields after the user finishes typing.
4. **Scroll Events:** Preventing frequent DOM updates while scrolling.

**Example:**

```
const search = (query) => console.log(`Searching for ${query}`);
const debouncedSearch = debounce(search, 300);

document.getElementById("searchInput").addEventListener("input", (e) => {
  debouncedSearch(e.target.value);
});
```

This example debounces the search function to avoid sending too many API requests.

## 86. What is throttling in JavaScript?

**Answer:**

1. **Throttling** ensures that a function is executed at most once in a specified time interval, regardless of how many times it is triggered.
2. It controls the rate of function execution, making it useful for events that fire frequently, such as scrolling, resizing, or mouse movements.
3. Improves performance by reducing the number of function calls during high-frequency events.
4. A timer is used to control the execution frequency of the function within the specified interval.
5. Throttling is ideal when consistent periodic updates are required, such as updating UI elements during scrolling.

**Example:**

```
function throttle(func, limit) {  
  let lastCall = 0;  
  return function (...args) {  
    const now = Date.now();  
    if (now - lastCall >= limit) {  
      lastCall = now;  
      func.apply(this, args);  
    }  
  };  
}  
  
// Example Usage  
const log = () => console.log("Throttled function executed");  
const throttledLog = throttle(log, 1000);  
  
window.addEventListener("scroll", throttledLog);
```

Here, the log function is called at most once every second while scrolling.

## 87. What are the practical use cases of throttling?

**Answer:**

1. **Scroll Events:** Reducing the frequency of updates to UI elements during scrolling.
2. **API Rate Limiting:** Preventing excessive API calls within a short time frame.
3. **Resize Events:** Adjusting layouts or dimensions while resizing the window.
4. **Button Clicks:** Preventing multiple clicks on a button in quick succession.

**Example:**

```
const handleScroll = () => console.log("Scrolled!");
```

```
const throttledScroll = throttle(handleScroll, 500);

window.addEventListener("scroll", throttledScroll);
```

This ensures that the handleScroll function executes at most once every 500ms during scrolling.

## 88. What is the difference between debouncing and throttling?

**Answer:**

Feature	Debouncing	Throttling
Purpose	Delays function execution until after a delay following the last event.	Limits the function execution rate to at most once in a given interval.
Execution	Executes the function once, after the event stops firing.	Executes the function at regular intervals while the event keeps firing.
Use Cases	Search bars, input fields, form validations.	Scroll events, resize events, API rate limiting.

**Example Comparison:**

```
// Debounce Example
const debouncedResize = debounce(() => console.log("Debounced Resize!"), 300);
window.addEventListener("resize", debouncedResize);

// Throttle Example
const throttledResize = throttle(() => console.log("Throttled Resize!"), 300);
window.addEventListener("resize", throttledResize);
```

## 89. Can you implement a combination of debouncing and throttling?

**Answer:**

Yes, you can combine debouncing and throttling to achieve a hybrid solution. For example, you may want to throttle events but ensure the function is called one final time after the events stop firing.

**Example:**

```
function debounceThrottle(func, delay, limit) {
  let lastCall = 0;
  let timeout;
  return function (...args) {
    const now = Date.now();
    clearTimeout(timeout);
    if (now - lastCall >= limit) {
      lastCall = now;
      func(...args);
    }
  };
}
```

```

        func.apply(this, args);
    }

    timeout = setTimeout(() => func.apply(this, args), delay);
};

}

// Example Usage
const log = () => console.log("Debounce + Throttle function executed");
const debouncedThrottledLog = debounceThrottle(log, 1000, 500);

window.addEventListener("resize", debouncedThrottledLog);

```

This ensures the log function executes at most once every 500ms and one final time after 1 second of inactivity.

## 90. Which one should you choose, debouncing or throttling?

**Answer:**

- **Choose Debouncing:** When you need to wait for the user to finish an action (e.g., typing in a search bar or resizing a window).
- **Choose Throttling:** When you need consistent execution over time, even while an action is continuously occurring (e.g., updating scroll positions or limiting API calls).

The choice depends on the specific requirements of your application.

## Modules:

### 91. What is the difference between named exports and default exports?

**Answer:**

1. **Named Exports:** Allow you to export multiple values from a module using their names. These must be imported using the exact same names. **Example:**

```

// Exporting
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// Importing
import { add, subtract } from "./math.js";

```

2. **Default Exports:** Allow you to export a single value or function. They can be imported with any name. **Example:**

```

// Exporting
export default (a, b) => a + b;

```

```
// Importing
import sum from "./math.js";
console.log(sum(2, 3)); // Output: 5
```

You can use both named and default exports in the same file.

## 92. What is the difference between ES Modules and CommonJS?

**Answer:**

Feature	ES Modules (ESM)	CommonJS (CJS)
<b>Syntax</b>	Uses import and export	Uses require and module.exports
<b>Support</b>	Native support in modern browsers and Node.js (from v12)	Primarily used in Node.js
<b>Static/Dynamic</b>	Statically analyzed (tree-shaking supported)	Dynamically loaded
<b>Default Export</b>	export default	module.exports =

Example of CommonJS:

```
// Exporting
module.exports = { greet };

// Importing
const { greet } = require("./module.js");
```

## 93. What is the purpose of the export \* from syntax?

**Answer:**

The export \* from syntax re-exports all exports from another module. It is commonly used to create aggregated modules that consolidate exports from multiple modules.

**Example:**

```
// moduleA.js
export const a = 1;

// moduleB.js
export const b = 2;

// index.js
export * from "./moduleA.js";
export * from "./moduleB.js";

// main.js
import { a, b } from "./index.js";
```

```
console.log(a, b); // Output: 1, 2
```

This approach simplifies importing multiple related modules.

#### 94. What are the advantages of using modules?

**Answer:**

1. **Code Reusability:** Modules allow you to write reusable, modular pieces of code.
2. **Encapsulation:** Variables and functions are scoped to the module, avoiding global namespace pollution.
3. **Maintainability:** Code is easier to maintain and debug when broken into smaller parts.
4. **Lazy Loading:** Modules can be dynamically imported to reduce the initial load time of applications.

```
// Lazy loading a module
document.getElementById("button").addEventListener("click", async () => {
  const { greet } = await import("./module.js");
  console.log(greet("Bob"));
});
```

#### 95. What are cookies in JavaScript?

**Answer:**

1. Cookies are small pieces of data stored on the user's browser by a website.
2. They are used to store information such as session data, user preferences, or tracking data.
3. Cookies are automatically sent back and forth between the client and server with every HTTP request.
4. They are essential for maintaining state in stateless HTTP connections.
5. Cookies can be set, accessed, or modified using JavaScript through the document.cookie property.

**Example:**

```
document.cookie = "username=Mahesh";
console.log(document.cookie); // Output: username=Mahesh
```

#### 96. How do you set a cookie in JavaScript?

**Answer:**

You can set a cookie using the document.cookie property. Each cookie is added as a string in the format name=value; followed by optional attributes like expires or path.

**Example:**

```
// Setting a cookie
document.cookie =
  "username=Mahesh; expires=Fri, 31 Jan 2025 23:59:59 GMT; path=/; secure";
```

## 97. What are the differences between session cookies and persistent cookies?

**Answer:**

**1. Session Cookies:**

- Do not have an expiration date or max-age specified.
- Stored in memory and are deleted when the browser is closed.

**2. Persistent Cookies:**

- Have an expires or max-age attribute specified.
- Stored on the user's disk and persist across browser sessions until they expire or are deleted.

### Example of Persistent Cookie:

```
document.cookie = "theme=dark; max-age=86400"; // Expires in 1 day
```

## local storage and session storage:

## 98. What is the difference between local storage and session storage?

**Answer:**

Local storage and session storage are part of the Web Storage API and allow storing key-value pairs in the browser. The key differences are:

**1. Lifetime:**

- **Local Storage:** Data persists indefinitely until explicitly cleared.
- **Session Storage:** Data is cleared when the page session ends (e.g., when the tab or browser is closed).

**2. Scope:**

- **Local Storage:** Shared across all tabs/windows from the same origin.
- **Session Storage:** Limited to the specific tab/window.

### Example:

```
// Local Storage
localStorage.setItem("username", "Mahesh");
console.log(localStorage.getItem("username")); // Output: Mahesh

// Session Storage
sessionStorage.setItem("sessionID", "12345");
console.log(sessionStorage.getItem("sessionID")); // Output: 12345
```

## 99. How do cookies differ from local storage and session storage?

**Answer:**

Feature	Cookies	Local Storage	Session Storage
Storage	4KB per cookie	5MB per origin	5MB per origin

Feature	Cookies	Local Storage	Session Storage
<b>Lifetime</b>	Depends on expires or max-age	Persistent until manually cleared	Cleared when the browser is closed
<b>Scope</b>	Sent with HTTP requests	Accessible only via JavaScript	Accessible only via JavaScript
<b>Security</b>	Vulnerable without HttpOnly	More secure, no automatic transfer	More secure, no automatic transfer

Cookies are best suited for server-client communication, while local and session storage are ideal for storing client-side data.

## 100. How do you store complex objects in local storage?

### Answer:

Since local storage only stores strings, you need to serialize objects into JSON strings using `JSON.stringify` and deserialize them using `JSON.parse`.

### Example:

```
const user = { name: "Mahesh", age: 25 };

// Storing an object
localStorage.setItem("user", JSON.stringify(user));

// Retrieving an object
const storedUser = JSON.parse(localStorage.getItem("user"));
console.log(storedUser.name); // Output: Mahesh
```

## 101. What are the limitations of local storage?

### Answer:

- Storage Limit:** Local storage has a limit of about **5MB** per origin in most browsers.
- String-Only Data:** Only strings can be stored, requiring serialization for non-string data.
- No Expiration:** Data persists indefinitely unless explicitly cleared.
- Security Risks:** Data is accessible to JavaScript, making it vulnerable to cross-site scripting (XSS) attacks.

## 102. How do you check the available storage capacity?

### Answer:

There is no direct method to check the available capacity of local or session storage, but you can calculate it by repeatedly adding data until an error occurs.

### Example:

```
function checkStorageLimit(storage) {
  let data = "x";
```

```

try {
  while (true) {
    storage.setItem("test", data);
    data += data; // Exponentially increase data size
  }
} catch (e) {
  console.log(`Approximate storage limit: ${data.length / 1024} KB`);
  storage.removeItem("test");
}
}

checkStorageLimit(localStorage);

```

### 103. When should you use local storage and session storage?

**Answer:**

**1. Use Local Storage:**

- When the data needs to persist across browser sessions.
- For non-sensitive data such as theme preferences, application settings, or caching small amounts of data.

**2. Use Session Storage:**

- For temporary data that is only needed during the user's session (e.g., form inputs, tab-specific states).

**Example:**

```

// Local Storage for theme preferences
localStorage.setItem("theme", "dark");

// Session Storage for temporary form data
sessionStorage.setItem(
  "draft",
  JSON.stringify({ title: "My Post", content: "..." })
);

```

**Fetch API:**

### 104. How does the Fetch API handle HTTP errors?

**Answer:**

The Fetch API only rejects a promise if a **network error** occurs. It does not reject for HTTP errors like 404 or 500. These must be handled manually by checking the `response.ok` property or the `response.status` code.

**Example:**

```

fetch("https://api.example.com/data")
  .then((response) => {

```

```

if (!response.ok) {
  throw new Error(`HTTP Error! Status: ${response.status}`);
}
return response.json();
})
.then((data) => console.log(data))
.catch((error) => console.error("Error:", error));

```

## 105. How can you include query parameters in a fetch request?

### Answer:

Query parameters can be added directly to the URL by appending a query string.

### Example:

```

const params = new URLSearchParams({
  search: "JavaScript",
  page: 1,
});

fetch(`https://api.example.com/data?${params}`)
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));

```

Here, URLSearchParams constructs the query string from an object

## 106. How do you handle JSON and non-JSON responses with the Fetch API?

### Answer:

You can use the Content-Type header in the response to determine how to handle the data. For example, you might process JSON responses with `.json()` and plain text responses with `.text()`.

### Example:

```

fetch("https://api.example.com/data")
  .then((response) => {
    const contentType = response.headers.get("Content-Type");
    if (contentType.includes("application/json")) {
      return response.json();
    } else if (contentType.includes("text/plain")) {
      return response.text();
    } else {
      throw new Error("Unsupported content type");
    }
  })
  .then((data) => console.log(data))

```

```
.catch((error) => console.error("Error:", error));
```

## 107. How can you use async/await with the Fetch API?

### Answer:

You can use async/await to simplify asynchronous fetch operations and handle errors using try-catch.

### Example:

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    if (!response.ok) {
      throw new Error(`HTTP Error! Status: ${response.status}`);
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}

fetchData();
```

## 108. How do you include headers in a fetch request?

### Answer:

You can include headers by specifying them in the headers property of the options object. Common headers include Authorization for tokens and Content-Type for specifying the request body format.

### Example:

```
fetch("https://api.example.com/data", {
  method: "GET",
  headers: {
    Authorization: "Bearer your-token",
    "Content-Type": "application/json",
  },
})
.then((response) => response.json())
.then((data) => console.log(data))
.catch((error) => console.error("Error:", error));
```

## Introduction of JavaScript, Datatypes and operators:

### 1. What will be the output of the following JavaScript code?

```
console.log(typeof foo);
var foo = function () {
  console.log("Hello, World!");
};
console.log(typeof foo);
```

#### Output:

```
undefined
function
```

#### Explanation:

- Before the function is assigned to foo, foo is declared but doesn't have a value—it's undefined.
- The first console.log checks the type of foo at this point, so it outputs "undefined".
- After assigning the function to foo, its type becomes "function".
- The second console.log reflects this change, outputting "function".

### 2. What will be the output of the following JavaScript code?

```
console.log([] + []);
console.log([] + { });
console.log({ } + []);
console.log({ } + { });
console.log(+true + "1" + false);
console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
```

#### Output:

```
"";
"[object Object]";
0;
("[object Object][object Object]");
("11false");
true;
false;
```

### **Explanation:**

1.  $[] + [] \rightarrow ""$

- o Arrays are converted to strings. An empty array becomes an empty string "".
  - o "" + "" results in "" (an empty string).
- 

2.  $[] + {} \rightarrow "[object Object]"$

- o The empty array [] converts to "" (empty string).
  - o The object {} converts to its string representation: "[object Object]".
  - o Concatenation: "" + "[object Object]" gives "[object Object]".
- 

3.  $\{ } + [] \rightarrow 0$

- o {} at the start is treated as a block (not an object).
  - o + [] coerces [] into an empty string "", and + "" converts it to 0.
  - o Result: 0.
- 

4.  $\{ } + \{ } \rightarrow "[object Object][object Object]"$

- o Both {} are treated as objects.
  - o Objects convert to strings: "[object Object]".
  - o Concatenation: "[object Object]" + "[object Object]" results in "[object Object][object Object]".
- 

5.  $+true + "1" + false \rightarrow "11false"$

- o +true converts true into 1 (using unary +).
  - o 1 + "1" results in "11" (number 1 coerces to string).
  - o "11" + false results in "11false" (false coerces to string).
- 

6.  $1 < 2 < 3 \rightarrow true$

- o JavaScript evaluates left-to-right.
  - o  $1 < 2$  is true.
  - o true is coerced to 1 in  $1 < 3$ , which evaluates to true.
- 

7.  $3 > 2 > 1 \rightarrow false$

- o Left-to-right evaluation.
- o  $3 > 2$  is true.
- o true is coerced to 1 in  $1 > 1$ , which evaluates to false.

### **3. What will be the output of the following JavaScript code ?**

```
console.log(+ "123");
console.log(+ null);
console.log(+ undefined);
console.log( + []);
console.log( + {});
```

**Output:**

```
123;
0;
NaN;
0;
NaN;
```

**Explanation:**

- +"123": Converts the string "123" to a number → 123.
- +null: Null coerces to 0 → 0.
- +undefined: Undefined cannot be converted to a number → NaN.
- +[]: An empty array coerces to an empty string, which converts to 0 → 0.
- +{}: Objects cannot be directly converted to numbers → NaN.

#### 4. What will be the output of the following JavaScript code ?

```
console.log("5" + 3 - 2);
console.log("5" - "3" + "2");
console.log("10" - "5" + 2 + "px");
```

**Output:**

```
51
22
7px
```

**Explanation:**

1. "5" + 3 - 2: "5" + 3 concatenates to "53". "53" - 2 coerces "53" to 53 → 51.
2. "5" - "3" + "2": "5" - "3" becomes 2. 2 + "2" concatenates to "22".
3. "10" - "5" + 2 + "px": "10" - "5" becomes 5. 5 + 2 is 7. 7 + "px" concatenates to "7px".

#### 5. What will be the output of the following JavaScript code ?

```
console.log(null == 0);
console.log(null >= 0);
console.log(null + []);
console.log(true + "1" == "true1");
```

**Output:**

```
false;
```

```
true;  
("null");  
false;
```

**Explanation:**

1. `null == 0`: Null only equals undefined (not numbers) → false.
2. `null >= 0`: Comparison converts null to 0, so `0 >= 0` → true.
3. `null + []`: null coerces to "null", and `+ []` to "". "null" + "" → "null".
4. `true + "1" === "true1"`: `true + "1"` results in "true1". Comparing with "true1" → false.

**6. What will be the output of the following JavaScript code ?**

```
console.log([] == ![]);  
console.log([1, 2] + [3, 4]);  
console.log({} == {});  
console.log("0" == false);
```

**Output:**

```
true;  
("1,23,4");  
false;  
true;
```

**Explanation:**

1. `[] == ![]`: `![]` is false, and `[] == false` (both coerced to "") → true.
2. `[1, 2] + [3, 4]`: Arrays are coerced to strings. "1,2" + "3,4" → "1,23,4".
3. `{ } == {}`: Objects are compared by reference, not value → false.
4. `"0" == false`: `false` coerces to 0, and `"0"` coerces to 0 → true.

**7. What will be the output of the following JavaScript code ?**

```
console.log(typeof null);  
console.log(null instanceof Object);  
console.log(typeof NaN);  
console.log(isNaN("abc" - 3));
```

**Output:**

```
object;  
false;  
number;  
true;
```

**Explanation:**

1. `typeof null`: Legacy behavior; `null` is treated as object.
2. `null instanceof Object`: `null` is not an instance of `Object` → false.
3. `typeof NaN`: `NaN` is a special value of type number → number.
4. `isNaN("abc" - 3)`: "`abc`" - 3 results in `NaN`, and `isNaN(NaN)` → true.

## 8. What will be the output of the following JavaScript code ?

```
console.log(null == undefined);
console.log(null === undefined);
console.log(!null);
console.log(Boolean({}));
```

### Output:

```
true;
false;
false;
true;
```

### Explanation:

1. `null == undefined`: Loose equality considers `null` and `undefined` as equal → true.
2. `null === undefined`: Strict equality requires matching types; `null` and `undefined` are different → false.
3. `!null`: `null` is falsy, so `!null` is false.
4. `Boolean({})`: Non-empty objects are always truthy → true.

## 9. What will be the output of the following JavaScript code ?

```
console.log(typeof 42n);
console.log(10n + 20n);
console.log(10n === 10);
console.log(typeof Symbol("id"));
```

### Output:

```
bigint;
30n;
false;
symbol;
```

### Explanation:

1. `typeof 42n`: The `n` suffix denotes a `BigInt`, and its type is "bigint".
2. `10n + 20n`: Arithmetic operations on `BigInt` values result in a `BigInt` → `30n`.
3. `10n === 10`: `BigInt` and `Number` are different types → false.
4. `typeof Symbol("id")`: Symbols are unique identifiers with type "symbol".

**10. What will be the output of the following JavaScript code ?**

```
console.log(3 || 0 && 2);
console.log(0 && 3 || 2);
console.log(0 ?? 5);
console.log(undefined ?? 5);
console.log(null ?? 0 ?? 10);
```

**Output:**

```
3;
2;
5;
5;
0;
```

**Explanation:**

1.  $3 \parallel 0 \&\& 2$ :  $\&\&$  first.  $0 \&\& 2 \rightarrow 0$ . Then  $3 \parallel 0 \rightarrow 3$ .
2.  $0 \&\& 3 \parallel 2$ :  $\&\&$  first.  $0 \&\& 3 \rightarrow 0$ . Then  $0 \parallel 2 \rightarrow 2$ .
3.  $0 ?? 5$ : Nullish coalescing returns the right-hand operand if the left is null or undefined.  $0$  is not nullish  $\rightarrow 0$ .
4.  $undefined ?? 5$ : Left operand is undefined. Result  $\rightarrow 5$ .
5.  $null ?? 0 ?? 10$ :  $null ?? 0 \rightarrow 0$ , then  $0 ?? 10 \rightarrow 0$ .

**conditional statements:**

**11. What will be the output of the following JavaScript code ?**

```
let x = 0;
if ((x = 5)) {
  console.log("x is truthy");
} else {
  console.log("x is falsy");
}
```

**Output:**

```
x is truthy
```

**Explanation:**

- $x = 5$  is an assignment, not a comparison.
- The condition evaluates to 5, which is truthy, so "x is truthy" is logged.

**12. What will be the output of the following JavaScript code ?**

```
let c = 0;
let d = null;
```

```
if (c ?? d) {  
    console.log("c or d is not nullish");  
} else {  
    console.log("Both are nullish");  
}
```

**Output:**

```
Both are nullish
```

**Explanation:**

- `c ?? d` evaluates to `c` since `c` is not null or undefined.
- `c` is 0, which is falsy, so the else block executes.

**switch and loops:**

**13. What will be the output of the following JavaScript code ?**

```
const num = 10;  
switch (true) {  
    case num < 5:  
        console.log("Less than 5");  
    case num < 15:  
        console.log("Less than 15");  
    default:  
        console.log("Default case");  
}
```

**Output:**

```
Less than 15  
Default case
```

**Explanation:**

- The first case fails, but `num < 15` evaluates to true.
- Without a break, it falls through to the default case.

**14. What will be the output of the following JavaScript code ?**

```
outer: for (let i = 0; i < 3; i++) {  
    for (let j = 0; j < 3; j++) {  
        if (i === 1 && j === 1) break outer;  
        console.log(i, j);  
    }  
}
```

**Output:**

```
0 0  
0 1  
0 2  
1 0  
1 1
```

**Explanation:**

- The break outer exits the outer loop when i === 1 and j === 1.
- All iterations before this condition are logged.

**15. What will be the output of the following JavaScript code ?**

```
const arr = [1, 2, 3];  
arr[5] = 10;  
console.log(arr);  
console.log(arr.length);
```

**Output:**

```
[1, 2, 3, <2 empty items>, 10]  
6
```

**Explanation:**

- Assigning arr[5] = 10 creates a sparse array with two empty slots.
- The length of the array becomes 6 due to the highest index (5) + 1.

**16. What will be the output of the following JavaScript code ?**

```
const arr = [1, [2, [3, [4, [5]]]]];  
const result = arr.flat(Infinity);  
console.log(result);
```

**Output:**

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

**Explanation:**

- flat(Infinity) flattens the array completely, regardless of depth, into a single-level array.

**17. What will be the output of the following JavaScript code ?**

```
const arr = [1, 2, 3];  
const mapResult = arr.map((x) => x * 2);  
const forEachResult = arr.forEach((x) => x * 2);  
console.log(mapResult);
```

```
console.log(forEachResult);
```

**Output:**

```
[2, 4, 6];  
undefined;
```

**Explanation:**

- map creates a new array by applying the callback function to each element.
- forEach does not return a new array; it only executes the callback for each element and returns undefined.

### 18. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3, 4, 5];  
const result = arr  
  .filter((x) => x % 2 === 0)  
  .map((x) => x * 2)  
  .reduce((acc, curr) => acc + curr, 0);  
console.log(result);
```

**Output:**

```
12;
```

**Explanation:**

- filter( $x \Rightarrow x \% 2 === 0$ ): Keeps even numbers  $\rightarrow [2, 4]$ .
- map( $x \Rightarrow x * 2$ ): Doubles each value  $\rightarrow [4, 8]$ .
- reduce( $((acc, curr) \Rightarrow acc + curr, 0)$ ): Sums the values  $\rightarrow 4 + 8 = 12$ .

### 19. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3, 4];  
arr.splice(1, 0, "a", "b");  
console.log(arr);  
arr.splice(-2, 1, "x");  
console.log(arr);
```

**Output:**

```
[1, "a", "b", 2, 3, 4]  
[1, "a", "b", 2, "x", 4]
```

**Explanation:**

1. splice(1, 0, "a", "b"): Adds "a" and "b" at index 1 without removing anything.
2. splice(-2, 1, "x"): At position -2 (second to last), removes 1 element (3) and inserts "x".

**20. What will be the output of the following JavaScript code ?**

```
const arr = [10, 5, 20, 15];
arr.sort((a, b) => b - a);
console.log(arr);
```

**Output:**

```
[20, 15, 10, 5];
```

**Explanation:**

- The custom comparator sorts the array in descending order:
  - $b - a$  ensures higher numbers come before lower numbers.

**21. What will be the output of the following JavaScript code ?**

```
const arr = [1, , 3];
console.log(arr.map((x) => x || 0));
console.log(arr.filter((x) => x !== undefined));
console.log(arr.length);
```

**Output:**

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

**Explanation:**

1. `map(x => x || 0)`: Maps empty slots to 0.
2. `filter(x => x !== undefined)`: Excludes undefined values (skips empty slots).
3. `.length`: Includes empty slots, so the length is 3.

**22. What will be the output of the following JavaScript code ?**

```
const arr = [10, 20, 30];
console.log(arr.some((x) => x > 25));
console.log(arr.every((x) => x > 25));
```

**Output:**

```
true;
false;
```

**Explanation:**

1. `some(x => x > 25)`: Returns true if at least one element satisfies the condition ( $30 > 25$ ).

- every( $x \Rightarrow x > 25$ ): Returns false because not all elements satisfy the condition.

### 23. What will be the output of the following JavaScript code ?

```
const arr = [1, 2, 3];
const reversed = arr.reverse();
console.log(reversed);
console.log(arr);
```

**Output:**

```
[3, 2, 1]
[3, 2, 1];
```

### Explanation:

- reverse reverses the array **in place** and also returns the reversed array.
- Both reversed and arr reference the same mutated array.

### 24. What will be the output of the following JavaScript code ?

```
const arr1 = Array.from("hello");
const arr2 = Array.of(1, 2, 3);
console.log(arr1);
console.log(arr2);
```

**Output:**

```
["h", "e", "l", "l", "o"]
[1, 2, 3]
```

### Explanation:

- Array.from("hello"): Creates an array from the string by splitting each character.
- Array.of(1, 2, 3): Creates an array from the given arguments.

### 25. What will be the output of the following JavaScript code ?

```
const arr = [1, -2, 3, -4];
const index = arr.findIndex((x) => x < 0);
console.log(index);
console.log(arr[index]);
```

**Output:**

```
1
-2
```

### Explanation:

- findIndex( $x \Rightarrow x < 0$ ): Returns the index of the first element less than 0 → 1.
- arr[index]: Accesses the element at index 1 → -2.

**26. What will be the output of the following JavaScript code ?**

```
function testArgs(a, b) {  
    console.log(arguments[0]);  
    console.log(arguments[1]);  
}  
testArgs(10, 20);
```

**Output:**

```
10;  
20;
```

**Explanation:**

- The arguments object holds all arguments passed to the function, even if parameters are not explicitly defined.

**27. What will be the output of the following JavaScript code ?**

```
const obj = { value: 10 };  
function getValue(multiplier) {  
    return this.value * multiplier;  
}  
console.log(getValue.call(obj, 2));  
console.log(getValue.apply(obj, [2]));  
const boundFn = getValue.bind(obj, 2);  
console.log(boundFn());
```

**Output:**

```
20;  
20;  
20;
```

**Explanation:**

- .call: Invokes the function with this set to obj and arguments provided individually.
- .apply: Invokes the function with this set to obj and arguments as an array.
- .bind: Returns a new function with this permanently set to obj and arguments pre-applied.

**30. What will be the output of the following JavaScript code ?**

```
const key = "dynamicKey";  
const obj = {  
    [key]: 42,  
};  
console.log(obj.dynamicKey);
```

**Output:**

```
42;
```

**Explanation:**

- Computed property [key] evaluates to "dynamicKey".
- The property dynamicKey is created with a value of 42.

**31. What will be the output of the following JavaScript code ?**

```
const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { ...obj1 };
obj2.b.c = 42;
console.log(obj1.b.c);
```

**Output:**

42

**Explanation:**

- The spread operator creates a **shallow copy**.
- Modifying obj2.b.c affects obj1.b.c because b is a reference.

**32. What will be the output of the following JavaScript code ?**

```
const obj = { x: 1 };
Object.freeze(obj);
obj.x = 42;
console.log(obj.x);
```

**Output:**

1;

**Explanation:**

- Object.freeze prevents modifications to the object.
- Attempts to modify x are ignored.

**33. What will be the output of the following JavaScript code ?**

```
const obj = Object.create({ a: 1 });
obj.b = 2;
console.log(obj.a);
console.log(obj.b);
console.log("a" in obj);
console.log(obj.hasOwnProperty("a"));
```

**Output:**

1;  
2;  
true;  
false;

**Explanation:**

- a is accessed through the prototype chain.
- b is directly defined on obj.
- "a" in obj checks for a in the object or its prototype.
- hasOwnProperty only checks properties directly on obj.

**DOM:**

**34. What will be the output of the following JavaScript code ?**

```
<!DOCTYPE html>
<html>
<body>
<div id="box" class="red"></div>
<script>
  const box = document.getElementById("box");
  box.classList.add("blue");
  box.classList.remove("red");
  console.log(box.className);
</script>
</body>
</html>
```

**Output:**

blue;

**Explanation:**

- classList.add("blue") adds the blue class.
- classList.remove("red") removes the red class.

**Event Handler:**

**35. What will be the output of the following JavaScript code ?**

```
<!DOCTYPE html>
<html>
<body>
<button id="btn">Click Me</button>
<script>
  const button = document.getElementById("btn");
  button.addEventListener("click", () => console.log("Clicked"), { once: true });
  button.click();
  button.click();
</script>
</body>
```

```
</html>
```

**Output:**

```
Clicked;
```

**Explanation:**

- The once: true option ensures the event listener is automatically removed after the first invocation.

**Event Loop:**

**36. What will be the output of the following JavaScript code ?**

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout");
}, 0);

Promise.resolve().then(() => {
  console.log("Promise");
});

console.log("End");
```

**Output:**

```
Start;
End;
Promise;
Timeout;
```

**Explanation:**

1. "Start" and "End" are logged synchronously.
2. Promise.then is a microtask and executes before the macrotask (setTimeout).
3. "Promise" is logged before "Timeout".

**37. What will be the output of the following JavaScript code ?**

```
console.log("1");

Promise.resolve().then(() => {
  console.log("2");
  Promise.resolve().then(() => {
    console.log("3");
  });
});

setTimeout(() => {
```

```
    console.log("4");
}, 0);
```

```
console.log("5");
```

**Output:**

```
1;
5;
2;
3;
4;
```

**Explanation:**

1. "C" is logged synchronously.
2. "A" is logged before the await.
3. "D" is logged synchronously while the microtask from await is queued.
4. "B" is logged when the microtask executes.

### 38. What will be the output of the following JavaScript code ?

```
setTimeout(() => console.log("Timeout"), 0);
setImmediate(() => console.log("Immediate"));
```

**Output:**

```
Immediate;
Timeout;
```

**Explanation:**

- In Node.js, setImmediate executes before setTimeout when both are scheduled simultaneously.

### 39. What will be the output of the following JavaScript code ?

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout");
}, 0);

const start = Date.now();
while (Date.now() - start < 1000) {}

console.log("End");
```

**Output:**

```
Start;
End;
```

**Timeout;**

**Explanation:**

1. The synchronous while loop blocks the event loop for 1 second.
2. "Timeout" is logged only after the loop ends and the event loop processes the macrotask.

**40. What will be the output of the following JavaScript code ?**

```
for (let i = 0; i < 3; i++) {  
    setTimeout(() => {  
        console.log(i);  
    }, 1000);  
}
```

**Output:**

```
0;  
1;  
2;
```

**Explanation:**

- let creates a block-scoped variable i for each iteration, so each setTimeout captures its own value.

**41. What will be the output of the following JavaScript code ?**

```
async function test() {  
    console.log("Start");  
    const res1 = await Promise.resolve("A");  
    console.log(res1);  
    const res2 = await Promise.resolve("B");  
    console.log(res2);  
}  
test();  
console.log("End");
```

**Output:**

```
Start;  
End;  
A;  
B;
```

**Explanation:**

1. "Start" is logged synchronously.
2. "End" is logged while the microtasks (await) are queued.
3. "A" and "B" are logged as the await resolves sequentially.

### **De-structuring:**

**42. What will be the output of the following JavaScript code ?**

```
let a = 5, b = 10;  
[a, b] = [b, a];  
console.log(a, b);
```

**Output:**

```
10 5
```

### **Explanation:**

- Destructuring swaps the values of a and b without using a temporary variable.

**43. What will be the output of the following JavaScript code ?**

```
const obj = { a: { b: { c: 42 } } };  
const { a: { b: { c } } } = obj;  
console.log(c);
```

**Output:**

```
42;
```

### **Explanation:**

- Nested destructuring extracts the deeply nested value c.

**44. What will be the output of the following JavaScript code ?**

```
const key = "name";  
const obj = { name: "Mahesh", age: 25 };  
const { [key]: value } = obj;  
console.log(value);
```

**Output:**

```
Mahesh;
```

### **Explanation:**

- Dynamic property names ([key]) allow destructuring using variables.

### **Prototype:**

**45. What will be the output of the following JavaScript code ?**

```
function Gadget() {}  
Gadget.prototype.type = "Generic";  
  
const gadget1 = new Gadget();
```

```
Gadget.prototype.type = "Updated";
```

```
console.log(gadget1.type);
```

**Output:**

```
Updated;
```

**Explanation:**

- Changing Gadget.prototype.type affects all instances that inherit from it unless they have an overridden type property.

**46. What will be the output of the following JavaScript code ?**

```
function Machine() {}  
const machine = new Machine();
```

```
console.log(Machine.prototype === machine.__proto__);
```

**Output:**

```
true;
```

**Explanation:**

- The prototype property of the constructor function (Machine) is the prototype of the instances (machine.\_\_proto\_\_).

**47. What will be the output of the following JavaScript code ?**

```
function Parent() {}  
Parent.prototype.greet = "Hello";
```

```
function Child() {}  
Child.prototype = Object.create(Parent.prototype);  
Child.prototype.constructor = Child;
```

```
const child = new Child();  
console.log(child.greet);  
console.log(child instanceof Parent);  
console.log(child.constructor === Child);
```

**Output:**

```
Hello  
true  
true
```

**Explanation:**

1. Child.prototype = Object.create(Parent.prototype) makes Child inherit from Parent.
2. instanceof verifies the inheritance chain.

3. constructor is explicitly set to Child.

**48. What will be the output of the following JavaScript code ?**

```
const arr = [1, 2, 3];
console.log(arr.__proto__ === Array.prototype);
console.log(Array.prototype.__proto__ === Object.prototype);
```

**Output:**

```
true;
true;
```

**Explanation:**

- Arrays inherit from Array.prototype, which in turn inherits from Object.prototype.

**49. What will be the output of the following JavaScript code ?**

```
class Calculator {
  static add(a, b) {
    return a + b;
  }
}

console.log(Calculator.add(5, 3));
const calc = new Calculator();
console.log(calc.add(2, 2));
```

**Output:**

```
8
TypeError: calc.add is not a function
```

**Explanation:**

- add is a static method, accessible via the class Calculator.add(5, 3).
- Static methods are not available on instances.
- Attempting to call calc.add(2, 2) results in a TypeError.

**50. What will be the output of the following JavaScript code ?**

```
class Rectangle {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
  get area() {
    return this.width * this.height;
  }
}
```

```
const rect = new Rectangle(5, 10);
console.log(rect.area);
rect.area = 100;
console.log(rect.area);
```

**Output:**

```
50;
50;
```

**Explanation:**

- The area property is a getter; it's computed from width and height.
- Attempting to set rect.area has no effect, as there is no setter defined.
- rect.area remains 50.

## 51. What will be the output of the following JavaScript code ?

```
const methodName = "greet";

class Greeter {
  [methodName]() {
    return "Hello!";
  }
}

const greeter = new Greeter();
console.log(greeter.greet());
```

**Output:**

```
Hello!
```

**Explanation:**

- Computed method names allow dynamic method names in classes.
- The method greet is defined using [methodName]().

## 52. What will be the output of the following JavaScript code ?

```
const p = new Person("John");
class Person {
  constructor(name) {
    this.name = name;
  }
}

console.log(p.name);
```

**Output:**

```
ReferenceError: Cannot access 'Person' before initialization
```

### **Explanation:**

- Unlike function declarations, class declarations are not hoisted.
- Attempting to create an instance before the class declaration results in a ReferenceError.

### **Arrow Functions, Closures, and Callbacks:**

#### **53. What will be the output of the following JavaScript code ?**

```
function counter() {  
    let count = 0;  
    return function () {  
        count++;  
        return count;  
    };  
}  
  
const increment = counter();  
console.log(increment());  
console.log(increment());  
console.log(increment());
```

#### **Output:**

```
1;  
2;  
3;
```

### **Explanation:**

- The inner function forms a **closure**, retaining access to the count variable in the outer function.
- Each call to increment() increments and returns the same count variable.

#### **54. What will be the output of the following JavaScript code ?**

```
function test() {  
    const arrow = () => console.log(arguments[0]);  
    arrow();  
}  
  
test(10);
```

#### **Output:**

```
10;
```

### **Explanation:**

- Arrow functions do not have their own arguments object; they inherit it from the enclosing regular function test.

**55. What will be the output of the following JavaScript code ?**

```
console.log("1");
setTimeOut(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
```

**Output:**

```
1;
4;
3;
2;
```

**Explanation:**

1. "1" and "4" are synchronous.
2. Promise.then (microtask) executes before setTimeout (macrotask).

**56. What will be the output of the following JavaScript code ?**

```
function outer() {
  const name = "Outer";
  const arrowFunc = () => {
    console.log(name);
  };
  arrowFunc();
}

outer();
```

**Output:**

```
Outer;
```

**Explanation:**

- Arrow functions have **lexical scope** and access the name variable from the enclosing outer function.

**57. What will be the output of the following JavaScript code ?**

```
for (var i = 0; i < 3; i++) {
  setTimeOut(() => console.log(i), 1000);
}
```

**Output:**

```
3;
3;
```

3;

**Explanation:**

- var is function-scoped, so the same i is referenced in each iteration.
- By the time setTimeout runs, i is 3.

**58. What will be the output of the following JavaScript code ?**

```
for (var i = 0; i < 3; i++) {  
  (function (j) {  
    setTimeout(() => console.log(j), 1000);  
  })(i);  
}
```

**Output:**

0;  
1;  
2;

**Explanation:**

- Using an IIFE with j creates a closure, capturing the correct value of i for each iteration.

**Debouncing and Throttling:**

**59. What will be the output of the following JavaScript code ?**

```
function debounce(func, delay) {  
  let timer;  
  return function (...args) {  
    clearTimeout(timer);  
    timer = setTimeout(() => func(...args), delay);  
  };  
}  
  
const search = debounce((query) => console.log("Searching for:", query), 500);  
  
search("A");  
search("AB");  
search("ABC");  
setTimeout(() => search("ABCD"), 600);
```

**Output:**

Searching for: ABC  
Searching for: ABCD

**Explanation:**

1. search("A"), search("AB"), and search("ABC") are called in quick succession.
2. The timer resets on each call, and only "ABC" is logged after 500ms.
3. The setTimeout after 600ms triggers a final call with "ABCD".

## 60. What will be the output of the following JavaScript code ?

```
function throttle(func, delay) {
  let lastCall = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= delay) {
      lastCall = now;
      func(...args);
    }
  };
}

const log = throttle(() => console.log("Throttled!"), 1000);

log();
setTimeout(log, 500);
setTimeout(log, 1000);
setTimeout(log, 2000);
```

### Output:

```
Throttled!
Throttled!
Throttled!
```

### Explanation:

1. The first call executes immediately.
2. The second call at 500ms is ignored because it's within the 1-second throttle delay.
3. The calls at 1000ms and 2000ms execute because the delay has passed.

## Modules, Cookies, Local Storage, and Session Storage:

### 61. What will be the output of the following JavaScript code ?

```
// module.js
export const greet = "Hello, World!";
export function sayHi(name) {
  return `Hi, ${name}`;
}

// main.js
import { greet, sayHi } from "./module.js";
console.log(greet);
```

```
console.log(sayHi("Mahesh"));
```

**Output:**

```
Hello, World!  
Hi, Mahesh
```

**Explanation:**

1. **export** allows you to export specific variables or functions.
2. **import** retrieves them in another file.
3. greet and sayHi are imported as named exports.

### 62. What will be the output of the following JavaScript code ?

```
document.cookie = "username=Mahesh; path=/";  
console.log(document.cookie);
```

**Output:**

```
username = Mahesh;
```

**Explanation:**

- Assigning a new value to the same cookie (username) updates it.

### 63. What will be the output of the following JavaScript code ?

```
sessionStorage.setItem("sessionKey", "active");  
console.log(sessionStorage.getItem("sessionKey"));  
sessionStorage.clear();  
console.log(sessionStorage.getItem("sessionKey"));
```

**Output:**

```
active;  
null;
```

**Explanation:**

1. sessionStorage persists only for the session.
2. clear removes all data from sessionStorage.

### 64. What will be the output of the following JavaScript code ?

```
localStorage.setItem("a", 1);  
localStorage.setItem("b", 2);  
  
for (let i = 0; i < localStorage.length; i++) {  
  const key = localStorage.key(i);  
  console.log(key, localStorage.getItem(key));
```

```
}
```

**Output:**

```
a 1  
b 2
```

**Explanation:**

- `localStorage.key(i)` retrieves the key at a specific index.

**65. What will be the output of the following JavaScript code ?**

```
document.cookie = "auth=true; path=/; max-age=3600";  
localStorage.setItem("theme", "dark");  
sessionStorage.setItem("sessionID", "123");  
  
console.log(document.cookie);  
console.log(localStorage.getItem("theme"));  
console.log(sessionStorage.getItem("sessionID"));
```

**Output:**

```
auth = true;  
dark;  
123;
```

**Explanation:**

- Cookies, `localStorage`, and `sessionStorage` can coexist to store different kinds of data.

**Conditional Statements, Switch Statements and Loops:**

**Task 1: Write a program to print all prime numbers between 1 and 20.**

**Solution:**

```
function isPrime(num) {  
    if (num <= 1) return false; // Numbers <= 1 are not prime  
    for (let i = 2; i <= Math.sqrt(num); i++) {  
        if (num % i === 0) return false; // If divisible by any number, it's not prime  
    }  
    return true; // If no divisors found, the number is prime  
}  
  
function printPrimes(n) {  
    for (let i = 1; i <= n; i++) {
```

```
if (isPrime(i)) {
    console.log(i); // Print the number if it's prime
}
}

printPrimes(20);
```

### Output:

```
2;
3;
5;
7;
11;
13;
17;
19;
```

## isPrime Function

### Breakdown of the Steps:

1. **Check if num <= 1:**
  - o Numbers less than or equal to 1 (like 0 and 1) are **not prime**.
  - o Hence, the function returns false.
2. **Optimize Prime Check Using Math.sqrt:**
  - o We loop from 2 to Math.sqrt(num).
  - o Why Math.sqrt(num)?
    - If a number has a factor larger than its square root, it must also have a smaller factor that has already been checked.
    - For example:
      - To check if 16 is prime, you only need to test divisors 2, 3, and 4 (since  $4 * 4 = 16$ ).
3. **Check for Divisors:**
  - o If the number is divisible by any i in the loop ( $\text{num \% i} === 0$ ), the function returns false because it has a factor other than 1 and itself.
4. **Return true:**
  - o If no divisors are found, the number is prime, and the function returns true.

## printPrimes Function

### Steps:

1. Use a for loop to iterate from 1 to n (in this case, 20).
2. For each number i, call the isPrime function.
3. If isPrime(i) returns true, print the number (console.log(i)).

## Execution of printPrimes(20)

1. Start with  $i = 1$ :
  - o  $\text{isPrime}(1) \rightarrow \text{false} \rightarrow \text{Not printed.}$
2. Move to  $i = 2$ :
  - o  $\text{isPrime}(2) \rightarrow \text{true} \rightarrow \text{Print } 2.$
3.  $i = 3$ :
  - o  $\text{isPrime}(3) \rightarrow \text{true} \rightarrow \text{Print } 3.$
4.  $i = 4$ :
  - o  $\text{isPrime}(4) \rightarrow \text{false} \text{ (divisible by 2)} \rightarrow \text{Not printed.}$
5. Continue this for all numbers up to 20.  
The prime numbers are **2, 3, 5, 7, 11, 13, 17, 19.**

## Task 2 : Write a program to print the Fibonacci sequence up to n terms using a while loop.

### Solution:

```
function fibonacci(n) {  
    let a = 0, // First term  
        b = 1, // Second term  
        nextTerm; // Variable to store the next term  
    let count = 0; // Counter for the loop  
  
    while (count < n) {  
        console.log(a); // Print the current term  
        nextTerm = a + b; // Calculate the next term  
        a = b; // Move 'b' to 'a'  
        b = nextTerm; // Move 'nextTerm' to 'b'  
        count++; // Increment the counter  
    }  
}  
  
fibonacci(7);
```

### Output:

```
0;  
1;  
1;  
2;  
3;  
5;  
8;
```

## How It Works

### 1. Initialize Values:

- o Start with  $a = 0$  (first term) and  $b = 1$  (second term).
- o  $\text{nextTerm}$  is used to calculate the next term in the sequence.
- o  $\text{count}$  keeps track of how many terms have been printed.

## 2. While Loop:

- The condition  $count < n$  ensures the loop runs until  $n$  terms are printed.
- Print the current value of  $a$  (the Fibonacci term).
- Calculate the **next term**:  $nextTerm = a + b$ .
- Update values:
  - Move  $b$  to  $a$ .
  - Move  $nextTerm$  to  $b$ .
- Increment the count.

## 3. Repeat:

- The loop continues until  $n$  terms are printed.

### Task 3 : Write a program to print the following pattern:

```
*  
* *  
* * *  
* * * *  
* * * * *
```

#### Solution:

```
function printPattern(n) {  
    for (let i = 1; i <= n; i++) {  
        let row = ""; // Initialize an empty string for each row  
        for (let j = 1; j <= i; j++) {  
            row += "* "; // Append a star and a space for each column  
        }  
        console.log(row); // Print the row  
    }  
}  
  
printPattern(5);
```

### How It Works

#### 1. Outer Loop (Rows):

- The **outer loop** runs from  $i = 1$  to  $n$ .
- Each iteration represents a new row.

#### 2. Inner Loop (Stars in Each Row):

- The **inner loop** runs from  $j = 1$  to  $i$  (the current row number).
- For each iteration of the inner loop, a star (\*) followed by a space () is appended to the row string.

#### 3. Print Row:

- After the inner loop completes, print the row string using `console.log`.

#### 4. Repeat for Next Row:

- The outer loop increments, and the process repeats for the next row

### Array:

### Task 4 : Write a function `removeDuplicates` that removes duplicate elements from an array.

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

**Solution:**

```
function removeDuplicates(arr) {  
  return [...new Set(arr)]; // Step 1: Convert the array to a Set, then spread it into a new array.  
}  
  
console.log(removeDuplicates([1, 2, 2, 3, 4, 4, 5])); // Output: [1, 2, 3, 4, 5]
```

### Step-by-Step Explanation

1. **Set:**
  - o A Set automatically removes duplicate values.
  - o Example: new Set([1, 2, 2, 3]) → {1, 2, 3}.
2. **Spread Operator:**
  - o Converts the Set back into an array.
  - o Example: [...new Set([1, 2, 2])] → [1, 2].

**Task 5 : Write a function flattenArray that takes a deeply nested array and returns a single flattened array.**

```
1, [2, [3, [4, 5]]]);  
// Output: [1, 2, 3, 4, 5]
```

**Solution:**

```
function flattenArray(arr) {  
  return arr.flat(Infinity); // Step 1: Use flat() with Infinity to flatten all levels  
}  
  
console.log(flattenArray([1, [2, [3, [4, 5]]]]));  
// Output: [1, 2, 3, 4, 5]
```

### Explanation

1. **flat(Infinity):**
  - o The flat() method flattens nested arrays by a specified depth.
  - o Passing Infinity ensures all levels of nesting are flattened.
2. **Return:**
  - o The function directly returns the flattened array.

**Task 6 : Write a function to find all pairs of numbers in an array that add up to a given target.**

**Solution:**

```
function findPairs(arr, target) {  
  let result = [];  
  let seen = new Set();
```

```

for (let num of arr) {
  let complement = target - num;
  if (seen.has(complement)) {
    result.push([complement, num]); // If complement exists, add the pair
  }
  seen.add(num); // Add the current number to the set
}
return result;
}

console.log(findPairs([2, 4, 3, 5, 7, 8, 1], 9));
// Output: [[5, 4], [7, 2], [8, 1]]

```

## Step-by-Step Explanation

1. **Initialize Storage:**
  - o Use a Set to keep track of visited numbers.
  - o Use an array result to store pairs.
2. **Calculate Complement:**
  - o For each number, calculate the complement: target - num.
3. **Check for Complement:**
  - o If the complement exists in the Set, it means we found a pair.
4. **Update Set:**
  - o Add the current number to the Set for future checks.
5. **Return Result:**
  - o Return the array of pairs.

**Task 7 : Write a function findMissingNumbers that takes an array of numbers in a sequence and returns the numbers that are missing from the sequence.**

```

findMissingNumbers([1, 2, 4, 6, 7]);
// Output: [3, 5]

```

```

findMissingNumbers([10, 12, 14]);
// Output: [11, 13]

```

### Solution:

```

function findMissingNumbers(arr) {
  const max = Math.max(...arr); // Step 1: Find the largest number
  const min = Math.min(...arr); // Step 2: Find the smallest number
  const fullRange = Array.from({ length: max - min + 1 }, (_, i) => i + min); // Step 3:
  Generate the full range
  return fullRange.filter((num) => !arr.includes(num)); // Step 4: Filter numbers not in the
  input array
}
console.log(findMissingNumbers([1, 2, 4, 6, 7])); // Output: [3, 5]

```

```
console.log(findMissingNumbers([10, 12, 14])); // Output: [11, 13]
```

## Step-by-Step Explanation

1. **Find Minimum and Maximum Values:**
  - o Use Math.min(...arr) and Math.max(...arr) to get the smallest and largest numbers in the input array.
2. **Generate the Full Range:**
  - o Use Array.from to create an array containing all numbers from min to max.
  - o Example: If min = 1 and max = 7, generate [1, 2, 3, 4, 5, 6, 7].
3. **Filter Missing Numbers:**
  - o Use filter() to keep only those numbers from the full range that are **not included** in the input array.
  - o Check membership using arr.includes(num).
4. **Return Result:**
  - o Return the array of missing numbers.

## Task 8 : Write a function to find the most frequent element in an array.

### Solution:

```
function mostFrequent(arr) {  
    let counts = {};  
    let maxElement = null;  
    let maxCount = 0;  
  
    arr.forEach((item) => {  
        counts[item] = (counts[item] || 0) + 1; // Update count  
        if (counts[item] > maxCount) {  
            maxCount = counts[item]; // Track the highest count  
            maxElement = item; // Track the most frequent element  
        }  
    });  
    return maxElement;  
}  
  
console.log(mostFrequent([1, 3, 2, 3, 4, 1, 3])); // Output: 3
```

## Step-by-Step Explanation

1. **Track Counts:**
  - o Use an object counts to store occurrences of each element.
2. **Check Most Frequent:**
  - o Track the element with the highest count using maxCount and maxElement.
3. **Update Values:**
  - o For each occurrence, compare the count to maxCount and update accordingly.
4. **Return Result:**
  - o Return the element with the highest count.

### Function:

**Task 9 : Write a function factorial that calculates the factorial of a given number using recursion.**

**Solution:**

```
function factorial(n) {  
    if (n === 0 || n === 1) return 1; // Base case: factorial(0) = 1 and factorial(1) = 1  
    return n * factorial(n - 1); // Recursive case  
}  
  
console.log(factorial(5)); // Output: 120
```

### Step-by-Step Explanation

1. **Base Case:**
  - o If n is 0 or 1, the factorial is 1.
2. **Recursive Case:**
  - o Multiply n by the factorial of n - 1.
3. **Execution:**
  - o The function calls itself with smaller values until it reaches the base case.

**Task 10 : Write a function fibonacci that generates the nth Fibonacci number using recursion**

**Solution:**

```
function fibonacci(n) {  
    if (n === 0) return 0; // Base case 1  
    if (n === 1) return 1; // Base case 2  
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case  
}  
  
console.log(fibonacci(6)); // Output: 8
```

### Step-by-Step Explanation

1. **Base Cases:**
  - o If n is 0, return 0.
  - o If n is 1, return 1.
2. **Recursive Case:**
  - o Sum the results of the n-1 and n-2 Fibonacci numbers.
3. **Execution:**
  - o Calls continue until base cases are reached, then results are summed up.

**Task 11 : Write a function objectToArray that converts an object into an array of [key, value] pairs.**

**Solution:**

```
function objectToArray(obj) {  
    return Object.entries(obj); // Convert object to array of [key, value] pairs  
}  
  
const obj = { a: 1, b: 2, c: 3 };  
console.log(objectToArray(obj));  
// Output: [["a", 1], ["b", 2], ["c", 3]]
```

**Step-by-Step Explanation**

1. **Object.entries:**
  - o Converts the object into an array where each element is a [key, value] pair.
2. **Return the Result:**
  - o The resulting array contains all key-value pairs.

**Task 12 : Write a function isEmptyObject that checks if an object is empty.**

**Solution:**

```
function isEmptyObject(obj) {  
    return Object.keys(obj).length === 0; // Check if the object has no keys  
}  
  
console.log(isEmptyObject({ })); // Output: true  
console.log(isEmptyObject({ a: 1 })); // Output: false
```

**Step-by-Step Explanation**

1. **Get Keys:**
  - o Use Object.keys(obj) to get an array of keys.
2. **Check Length:**
  - o If the array length is 0, the object is empty.

[Event Handlers, Event Propagation, and Event Delegation:](#)

**Task 13 : Write a function toggleClassOnClick that adds or removes a CSS class when a button is clicked.**

**Solution:**

```
function toggleClassOnClick(buttonId, targetId, className) {  
    const button = document.getElementById(buttonId);  
    const target = document.getElementById(targetId);
```

```

button.addEventListener("click", () => {
  target.classList.toggle(className); // Step 3: Toggle the class
});
}

toggleClassOnClick("toggleButton", "myDiv", "highlight");

```

## Step-by-Step Explanation

1. **Select Elements:**
  - o Use document.getElementById to select the button and target element.
2. **Add an Event Listener:**
  - o Use .addEventListener("click") to listen for click events on the button.
3. **Toggle Class:**
  - o Use classList.toggle(className) to add or remove the class.

**Task 14 : Write a function animateElement that moves an element 10px to the right every 100ms.**

### Solution:

```

function animateElement(id) {
  const element = document.getElementById(id); // Step 1: Select the element by ID
  let position = 0;

  const interval = setInterval(() => {
    position += 10; // Step 2: Increment position
    element.style.transform = `translateX(${position}px)`; // Step 3: Update position
    if (position >= 100) {
      clearInterval(interval); // Step 4: Stop animation
    }
  }, 100);
}

animateElement("myDiv");

```

## Step-by-Step Explanation

1. **Select the Element:**
  - o Use document.getElementById(id) to find the element.
2. **Initialize Position:**
  - o Use a variable position to track the current position.
3. **Animate Using setInterval:**
  - o Update the position every 100ms using style.transform.
4. **Stop the Animation:**
  - o Use clearInterval() when the position reaches 100px.

**Task 15 : Write a function logEventPhases that adds event listeners in both the capturing and bubbling phases to demonstrate event propagation.**

**Solution:**

```
function logEventPhases() {  
  const parent = document.getElementById("parent");  
  const child = document.getElementById("child");  
  
  parent.addEventListener("click", () => {  
    console.log("Parent - Bubble Phase");  
  });  
  
  parent.addEventListener(  
    "click",  
    () => {  
      console.log("Parent - Capture Phase");  
    },  
    true // Enable capture phase  
  );  
  
  child.addEventListener("click", (event) => {  
    console.log("Child Clicked");  
  });  
}  
  
logEventPhases();
```

### Step-by-Step Explanation

1. **Capture Phase:**
  - o Add an event listener to parent with the third argument as true to enable capturing.
2. **Bubble Phase:**
  - o Add a standard event listener to parent (default is bubbling).
3. **Trigger Child Event:**
  - o Click on the child to see how the event flows through the DOM.

**Task 16 : Write a program to demonstrate the execution order of microtasks and macrotasks.**

**Solution:**

```
console.log("Start");  
  
setTimeout(() => {  
  console.log("Macrotask");  
}, 0);
```

```
Promise.resolve().then(() => {
  console.log("Microtask");
});
```

```
console.log("End");
```

**Output:**

```
Start;
End;
Microtask;
Macrotask;
```

### Step-by-Step Explanation:

#### 1. Synchronous Code:

- o Logs "Start" and "End" first.

#### 2. Microtask:

- o The Promise callback executes before the macrotask because microtasks have higher priority.

#### 3. Macrotask:

- o The setTimeout callback is executed after the microtask.

## Task 17 : Write a function to demonstrate the difference between setImmediate (Node.js) and setTimeout.

### Solution:

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout");
}, 0);

setImmediate(() => {
  console.log("Immediate");
});
console.log("End");
```

**Output:**

```
Start;
End;
Immediate;
Timeout;
```

### Step-by-Step Explanation:

#### 1. Synchronous Code:

- o Logs "Start" and "End".

#### 2. setImmediate:

- o Executes before setTimeout because setImmediate is added to the **check phase**, while setTimeout is in the **timer phase**.

## Spread and Rest:

**Task 18 : Create a shallow copy of an object or array using the spread operator. Demonstrate how changes in nested properties affect the copied version.**

### **Solution:**

```
// Shallow Copy of an Object
const originalObject = { a: 1, b: { c: 2 } };
const shallowCopyObject = { ...originalObject };

shallowCopyObject.b.c = 99;

console.log(originalObject); // Output: { a: 1, b: { c: 2 } }
console.log(shallowCopyObject); // Output: { a: 1, b: { c: 99 } }

// Shallow Copy of an Array
const originalArray = [1, [2, 3]];
const shallowCopyArray = [...originalArray];

shallowCopyArray[1][0] = 99;

console.log(originalArray); // Output: [1, [2, 3]]
console.log(shallowCopyArray); // Output: [1, [99, 3]]
```

### **Explanation**

1. **Shallow Copy:**
  - o Copies only the first level of the object or array.
  - o Nested objects or arrays are still **referenced**, meaning changes in the nested structure will affect both the original and the copy.
2. **Spread Operator:**
  - o Expands the properties of the object or array into a new container.

**Task 19 : Create a deep copy of an object or array so that changes in nested properties do not affect the original.**

### **Solution:**

```
const originalObject = { a: 1, b: { c: 2 } };
const deepCopyObject = JSON.parse(JSON.stringify(originalObject));

deepCopyObject.b.c = 99;

console.log(originalObject); // Output: { a: 1, b: { c: 2 } }
console.log(deepCopyObject); // Output: { a: 1, b: { c: 99 } }
```

### **Explanation**

1. **Deep Copy:**

- Creates an entirely new copy of the object or array, including all nested properties.
  - Modifications to the copied object or array do not affect the original.
2. **JSON Method:**
- `JSON.stringify()` converts the object to a string.
  - `JSON.parse()` converts the string back to a new object.
  - Limitation: It does not handle functions or undefined.

## Task 20 : Highlight the differences between shallow copy and deep copy with examples.

### Solution:

```
const original = { a: 1, b: { c: 2 } };

// Shallow Copy
const shallowCopy = { ...original };
shallowCopy.b.c = 99;

console.log("Shallow Copy:");
console.log("Original:", original); // Output: { a: 1, b: { c: 99 } }
console.log("Copy:", shallowCopy); // Output: { a: 1, b: { c: 99 } }

// Deep Copy
const deepCopy = JSON.parse(JSON.stringify(original));
deepCopy.b.c = 42;

console.log("Deep Copy:");
console.log("Original:", original); // Output: { a: 1, b: { c: 99 } }
console.log("Copy:", deepCopy); // Output: { a: 1, b: { c: 42 } }
```

### Creating the Shallow Copy:

- The **spread operator** (`{ ...original }`) creates a **shallow copy** of the original object.
- A shallow copy only duplicates the **first level** of properties.
  - The property `a` is copied as a new value because it's a primitive.
  - The property `b` (a nested object) is **referenced**, not duplicated.

### Modifying the Nested Object:

- When `shallowCopy.b.c = 99` is executed, it modifies the **same nested object (b)** that exists in the original object.
- This happens because `b` is **shared between the original and the shallow copy**.

Both the original and `shallowCopy` objects reflect the change to `b.c`, showing `c: 99`.

### Creating the Deep Copy:

- The `JSON.stringify(original)` converts the original object into a JSON string:  
`{"a":1,"b":{"c":2}}`.
- The `JSON.parse()` then parses the JSON string back into a **new object**.

- This creates a **deep copy** of the original object, duplicating all levels of nested properties.

### **Modifying the Nested Object:**

- When `deepCopy.b.c = 42` is executed, it modifies the `b` object in the `deepCopy`, which is **independent** of the `b` object in the original.
- Changes to `deepCopy` do not affect the original.

The original object remains unchanged with `b: { c: 99 }`.

The `deepCopy` has its own `b` object with `c: 42`.

### **Destructuring:**

#### **Task 21 : Given an object representing a user**

```
const user = {
  id: 1,
  details: {
    userName: "Mahesh",
    age: 25,
  },
  address: {
    city: "HYD",
    zip: 10001,
  },
};
```

#### **You need to:**

1. Extract the `userName` property from the `details` object.
2. Rename the extracted `userName` to `name`.
3. Extract the `city` property from the `address` object.

#### **Solution:**

```
const { details: { userName: name }, address: { city } } = user;
console.log(name); // Output: Mahesh
console.log(city); // Output: HYD
```

#### **Explanation**

Here's the destructuring explained step by step:

1. **Syntax:**

```
const { details: { userName: name }, address: { city }, } = user;
```

- o { details: { userName: name } }:
  - Access the details object, extract userName, and rename it to name.
- o { address: { city } }:
  - Access the address object and extract city.

## 2. Final Variables:

- o name holds the value "Mahesh".
- o city holds the value "HYD".

### Task 22 : : Given an object

```
const product = {  
  id: 101,  
  name: "Laptop",  
};
```

**Extract the name property.**

**Extract the stock property, but if it's missing, assign it a default value of 0.**

#### Solution:

```
const { name, stock = 0 } = product;  
  
console.log(name); // Output: Laptop  
console.log(stock); // Output: 0
```

Explanation:

- **Default Value:**
  - o stock = 0: If stock doesn't exist in the object, it will default to 0.

### Task 23 : Given three variables:

```
let a = 1,  
  b = 2,  
  c = 3;
```

**Swap their values such that a = 3, b = 1, and c = 2.**

#### Solution:

```
[a, b, c] = [c, a, b];  
  
console.log(a); // Output: 3  
console.log(b); // Output: 1  
console.log(c); // Output: 2
```

Explanation:

1. **Array Destructuring:**

- `[a, b, c] = [c, a, b]`: Creates a new array with values rearranged and assigns them back to a, b, and c.

### Task 24 : Given a nested array:

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

**Extract the first element of each sub-array.**

#### Solution:

```
const [[a], [b], [c]] = matrix;

console.log(a); // Output: 1
console.log(b); // Output: 3
console.log(c); // Output: 5
```

Explanation:

#### 1. Nested Destructuring:

- Use `[a], [b], [c]` to destructure each sub-array and extract its first element.

## HOF:

### Task 25 :

Write a function `multiplyBy` that:

- Accepts a number `factor`.
- Returns a new function that multiplies its input by `factor`.

#### Solution:

```
function multiplyBy(factor) {
  return (num) => num * factor;
}

const double = multiplyBy(2);
const triple = multiplyBy(3);

console.log(double(5)); // Output: 10
console.log(triple(5)); // Output: 15
```

Explanation:

1. **Higher-Order Function:**
  - o multiplyBy accepts factor and returns a function.
2. **Reusable Functions:**
  - o double and triple are created by passing specific factor values to multiplyBy.

### **Advanced Array methods:**

#### **Task 26 : From an array of numbers**

```
const numbers = [1, 2, 3, 4, 5, 6];  
create a new array of squares for even numbers only.
```

**Solution:**

```
const numbers = [1, 2, 3, 4, 5, 6];  
const evenSquares = numbers  
.filter((num) => num % 2 === 0)  
.map((num) => num * num);  
  
console.log(evenSquares); // Output: [4, 16, 36]
```

Explanation:

1. filter:
  - o Filters the array to include only even numbers [2, 4, 6].
2. map:
  - o Squares each number in the filtered array.

#### **Task 27 : Given an array of strings:**

```
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];
```

**Count how many times each fruit appears.**

**Solution:**

```
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];  
const counts = fruits.reduce((acc, fruit) => {  
  acc[fruit] = (acc[fruit] || 0) + 1;  
  return acc;  
}, {});
```

```
console.log(counts); // Output: { apple: 3, banana: 2, orange: 1 }
```

Explanation:

1. **reduce Function:**
  - o Uses an object (acc) to store counts for each fruit.
  - o Updates the count for each fruit: (acc[fruit] || 0) + 1.
2. **Output:**
  - o An object { apple: 3, banana: 2, orange: 1 }.

### Task 28 : Given a nested array:

```
const nestedArray = [  
  [1, 2],  
  [3, 4],  
  [5, 6],  
];
```

**Flatten it into a single array.**

Solution:

```
const nestedArray = [  
  [1, 2],  
  [3, 4],  
  [5, 6],  
];  
const flatArray = nestedArray.reduce((acc, arr) => acc.concat(arr), []);  
  
console.log(flatArray); // Output: [1, 2, 3, 4, 5, 6]
```

Explanation:

1. **reduce Function:**
  - o Combines each sub-array into the accumulator (acc).
  - o Uses concat to merge arrays.
2. **Output:**
  - o A flattened array [1, 2, 3, 4, 5, 6].

### Task 29 : Given an array with duplicates:

```
const numbers = [1, 2, 3, 2, 4, 1, 5];
```

- **Create a new array with unique numbers.**

**Solution:**

```
const numbers = [1, 2, 3, 2, 4, 1, 5];
const uniqueNumbers = numbers.filter(
  (num, index, arr) => arr.indexOf(num) === index
);

console.log(uniqueNumbers); // Output: [1, 2, 3, 4, 5]
```

Explanation:

1. **filter Function:**
  - o Keeps only the first occurrence of each number.
2. **indexOf:**
  - o Returns the index of the first occurrence of a number.
3. **Output:**
  - o A new array [1, 2, 3, 4, 5].

**Task 30 : Given an array of objects:**

```
const employees = [
  { name: "Mahesh", age: 30 },
  { name: "Hema", age: 25 },
  { name: "TATA", age: 35 },
];
```

**Sort the array by age in ascending order.**

**Solution:**

```
employees.sort((a, b) => a.age - b.age);

console.log(employees);
// Output: [
//   { name: "Hema", age: 25 },
//   { name: "Mahesh", age: 30 },
//   { name: "TATA", age: 35 }
// ]
```

Explanation:

1. **sort Method:**
  - o Compares two elements a and b based on their age property.
  - o a.age - b.age ensures ascending order.
2. **Result:**
  - o The array is sorted from the youngest to the oldest.

### Task 31 : Given an array of objects:

```
const students = [
  { name: "Mahesh", grade: "A" },
  { name: "Hema", grade: "B" },
  { name: "TATA", grade: "A" },
  { name: "Sudha", grade: "C" },
];
```

**Group the students by their grade.**

#### Solution:

```
const groupedByGrade = students.reduce((acc, student) => {
  const { grade } = student;
  acc[grade] = acc[grade] || [];
  acc[grade].push(student);
  return acc;
}, {});

console.log(groupedByGrade);
// Output: {
//   A: [{ name: "Mahesh", grade: "A" }, { name: "TATA", grade: "A" }],
//   B: [{ name: "Hema", grade: "B" }],
//   C: [{ name: "Sudha", grade: "C" }]
// }
```

Explanation:

1. **reduce:**
  - o Iterates over the array, adding each student to their respective grade group in the accumulator (acc).
2. **Grouping:**
  - o `acc[grade] = acc[grade] || []` initializes the group if it doesn't exist.
3. **Result:**
  - o A grouped object by grade.

### Task 32 : Given an array of objects:

```
const data = [
  { id: 1, name: "Sudha" },
  { id: 2, name: "Mahesh" },
  { id: 1, name: "Sudha" },
  { id: 3, name: "Hema" },
];
```

**Remove duplicates based on the id.**

#### Solution:

```
const uniqueData = data.filter(
  (obj, index, self) => index === self.findIndex((el) => el.id === obj.id)
```

```

);
console.log(uniqueData);
// Output: [
// { id: 1, name: "Sudha" },
// { id: 2, name: "Mahesh" },
// { id: 3, name: "Hema" }
// ]

```

Explanation:

1. filter:
  - o Filters elements by comparing their id with the index of the first occurrence in the array.
2. findIndex:
  - o Ensures only the first instance of each id is included.

### Task 33 : Given two arrays:

```

const array1 = [
  { id: 1, name: "Hema" },
  { id: 2, name: "Mahesh" },
];
const array2 = [
  { id: 1, age: 25 },
  { id: 2, age: 30 },
];

```

Merge them into a single array of objects by id.

Solution:

```

const mergedArray = array1.map((item) => ({
  ...item,
  ...array2.find((el) => el.id === item.id),
}));

console.log(mergedArray);
// Output: [
// { id: 1, name: "Hema", age: 25 },
// { id: 2, name: "Mahesh", age: 30 }
// ]

```

Explanation:

1. map:
  - o Iterates over array1 and creates a new object for each element.
2. find:
  - o Finds the matching object in array2 by id.

### 3. Spread Operator:

- o Combines properties from both objects.

#### Task 34 : Given an array of objects

```
const items = [  
  { id: 1, value: 10 },  
  { id: 2, value: 50 },  
  { id: 3, value: 30 },  
];
```

**Find the object with the maximum value.**

#### Solution:

```
const maxValueItem = items.reduce((max, item) =>  
  item.value > max.value ? item : max  
);  
  
console.log(maxValueItem); // Output: { id: 2, value: 50 }
```

Explanation:

#### 1. reduce:

- o Compares each object's value with the current maximum and updates the accumulator if needed.

#### 2. Result:

- o The object with the highest value is { id: 2, value: 50 }.

#### Task 35 : Given an array of objects:

```
const products = [  
  { name: "Laptop", price: 1000, category: "Electronics" },  
  { name: "Shirt", price: 50, category: "Clothing" },  
  { name: "Phone", price: 800, category: "Electronics" },  
  { name: "Pants", price: 60, category: "Clothing" },  
];
```

**Filter only products in the "Electronics" category.**

**Extract their price.**

**Calculate the total price of all filtered products.**

#### Solution:

```
const totalElectronicsPrice = products  
.filter((product) => product.category === "Electronics") // Step 1: Filter  
.map((product) => product.price) // Step 2: Extract price  
.reduce((total, price) => total + price, 0); // Step 3: Sum prices
```

```
console.log(totalElectronicsPrice); // Output: 1800
```

### Explanation:

1. filter:
  - o Selects only products in the "Electronics" category.
  - o Result: [ { name: "Laptop", price: 1000 }, { name: "Phone", price: 800 } ].
2. map:
  - o Extracts the price property.
  - o Result: [1000, 800].
3. reduce:
  - o Sums the prices.
  - o Result: 1800.

### Task 36: given an array of numbers:

```
const numbers = [1, 2, 3, 4, 5, 6];
```

**Filter out odd numbers.**

**Square the remaining numbers.**

**Calculate their sum.**

### Solution:

```
const result = numbers
  .filter((num) => num % 2 === 0) // Step 1: Filter evens
  .map((num) => num * num) // Step 2: Square them
  .reduce((sum, num) => sum + num, 0); // Step 3: Sum them

console.log(result); // Output: 56
```

### Explanation:

1. filter:
  - o Keeps only even numbers [2, 4, 6].
2. map:
  - o Squares each even number [4, 16, 36].
3. reduce:
  - o Sums the squared numbers 56.

### Task 37: given an array of objects:

```
const students = [
  { name: "Hema", marks: [85, 90, 80] },
  { name: "Mahesh", marks: [70, 75, 78] },
  { name: "Sudha", marks: [88, 92, 86] },
];
```

**Calculate the average marks of all students.**

**Solution:**

```
const averageMarks = students
    .map((student) => student.marks.reduce((a, b) => a + b, 0) / student.marks.length) // Step 1:
Calculate each student's average
    .reduce((acc, avg) => acc + avg, 0) / students.length; // Step 2: Find the overall average

console.log(averageMarks); // Output: 83.5
```

**Explanation:****map:**

- Calculates the average marks for each student.
- Result: [85, 74.33, 88.67].

**reduce:**

- Sums the averages and divides by the total number of students.
- Result: 83.5.

**Task 38: given an array of products**

```
const products = [
  { name: "Laptop", category: "Electronics" },
  { name: "Shirt", category: "Clothing" },
  { name: "Phone", category: "Electronics" },
  { name: "Pants", category: "Clothing" },
  { name: "Fridge", category: "Appliances" },
];
```

**Extract all unique categories.**

**Count how many products belong to each category.**

**Solution:**

```
const categoryCounts = products
    .map((product) => product.category) // Step 1: Extract categories
    .reduce((acc, category) => {
      acc[category] = (acc[category] || 0) + 1;
      return acc;
    }, {});

console.log(categoryCounts);
// Output: { Electronics: 2, Clothing: 2, Appliances: 1 }
```

**Explanation:****map:**

- Extracts the category property from each product.
- Result: ["Electronics", "Clothing", "Electronics", "Clothing", "Appliances"].

**reduce:**

- Counts occurrences of each category.
- Result: { Electronics: 2, Clothing: 2, Appliances: 1 }.

### Task 39: given an array of products

```
const products = [
  { id: 1, name: "Laptop", price: 1000, category: "Electronics" },
  { id: 2, name: "Shirt", price: 50, category: "Clothing" },
  { id: 3, name: "Phone", price: 800, category: "Electronics" },
  { id: 4, name: "Pants", price: 60, category: "Clothing" },
];
```

**Filter out products in the "Electronics" category.**

**Add a discountedPrice (10% off).**

**Calculate the total discountedPrice of these products.**

**Solution:**

```
const totalDiscountedPrice = products
  .filter((product) => product.category === "Electronics") // Step 1: Filter
  .map((product) => ({ ...product, discountedPrice: product.price * 0.9 })) // Step 2: Add
  discountedPrice
  .reduce((total, product) => total + product.discountedPrice, 0); // Step 3: Calculate total

console.log(totalDiscountedPrice); // Output: 1620
```

**Explanation:**

1. filter:
  - Filters products in the "Electronics" category.
  - Result: [{ id: 1, price: 1000 }, { id: 3, price: 800 }].
2. map:
  - Adds a discountedPrice (10% off).
  - Result: [{ id: 1, discountedPrice: 900 }, { id: 3, discountedPrice: 720 }].
3. reduce:
  - Sums up the discountedPrice.
  - Result: 1620.

### Task 40: given an array of orders, each containing a list of purchased products:

```
const orders = [
  { id: 1, products: ["Laptop", "Phone"] },
  { id: 2, products: ["Shirt", "Phone"] },
  { id: 3, products: ["Laptop", "Pants"] },
];
```

**Flatten the list of all products.**

**Remove duplicate products.**

**Solution:**

```
const uniqueProducts = orders
  .flatMap(order => order.products) // Step 1: Flatten product lists
  .filter((product, index, self) => self.indexOf(product) === index); // Step 2: Remove
  duplicates

console.log(uniqueProducts);
// Output: ["Laptop", "Phone", "Shirt", "Pants"]
```

**Explanation:****flatMap:**

- Flattens all product arrays into a single array.
- Result: ["Laptop", "Phone", "Shirt", "Phone", "Laptop", "Pants"].

**filter:**

- Removes duplicates using indexOf.
- Result: ["Laptop", "Phone", "Shirt", "Pants"].

**Task 41: given an array of students:**

```
const students = [
  { id: 1, name: "Hema", marks: 85 },
  { id: 2, name: "Mahesh", marks: 72 },
  { id: 3, name: "Sudha", marks: 60 },
  { id: 4, name: "TATA", marks: 92 },
];
```

**Check if any student failed (marks < 40).**

**Verify if all students passed (marks >= 60).**

**Calculate the average marks.**

**Solution:**

```
const anyFailed = students.some((student) => student.marks < 40); // Step 1: Check if any
failed
const allPassed = students.every((student) => student.marks >= 60); // Step 2: Check if all
passed
const averageMarks =
  students.reduce((total, student) => total + student.marks, 0) /
  students.length; // Step 3: Calculate average

console.log(anyFailed); // Output: false
console.log(allPassed); // Output: true
console.log(averageMarks); // Output: 77.25
```

**Explanation:****some:**

- Checks if any student failed (marks < 40).

**every:**

- Verifies if all students passed (marks  $\geq 60$ ).

**reduce:**

- Calculates the total marks and divides by the number of students.

### Cookies, Local Storage, and Session Storage:

**Task 42: to store a user's login status (true) in session storage and remove it after the session ends.**

**Solution:**

```
// Store login status
sessionStorage.setItem("isLoggedIn", "true");

// Retrieve and check login status
const isLoggedIn = sessionStorage.getItem("isLoggedIn") === "true";
console.log(isLoggedIn); // Output: true

// Remove login status
sessionStorage.removeItem("isLoggedIn");
console.log(sessionStorage.getItem("isLoggedIn")); // Output: null
```

**Explanation:**

**Session-Based Storage:**

- `sessionStorage.setItem(key, value)` stores data for the current session only.

**Remove Data:**

- `sessionStorage.removeItem(key)` deletes the stored key-value pair.

**Task 43: Create a cookie named "username" with the value "Mahesh", and set it to expire in 7 days.**

**Solution:**

```
// Set a cookie
document.cookie = "username=Mahesh; max-age=" + 7 * 24 * 60 * 60;

// Read cookies
console.log(document.cookie); // Output: "username=Mahesh"
```

**Explanation:**

### **Setting a Cookie:**

- Use document.cookie to set the cookie. The max-age specifies the expiration time in seconds (7 days).

### **Reading Cookies:**

- document.cookie retrieves all cookies as a single string.

**Task 44: Create a cookie consent banner that stores the user's consent in cookies and hides the banner once consent is given.**

### **Solution:**

```
// Check for consent cookie
if (!document.cookie.includes("consent=true")) {
    console.log("Show consent banner");

    // Simulate user giving consent
    document.cookie = "consent=true; max-age=" + 365 * 24 * 60 * 60; // 1 year
    console.log("Consent given");
} else {
    console.log("Consent already given");
}
```

### **Explanation:**

#### **Cookie Check:**

- Check if the "consent=true" cookie exists.

#### **Set Cookie:**

- If not, set the cookie when consent is given.

**Task 45: Store data in local storage with an expiry time. If the expiry has passed, remove the data.**

### **Solution:**

```
// Store data with expiry
const setWithExpiry = (key, value, ttl) => {
    const now = new Date();
    const item = {
        value,
        expiry: now.getTime() + ttl,
    };
    localStorage.setItem(key, JSON.stringify(item));
};

// Retrieve data with expiry check
const getWithExpiry = (key) => {
```

```

const itemStr = localStorage.getItem(key);
if (!itemStr) return null;

const item = JSON.parse(itemStr);
const now = new Date();

if (now.getTime() > item.expiry) {
  localStorage.removeItem(key);
  return null;
}
return item.value;
};

// Example usage
setWithExpiry("token", "abc123", 5000); // Expires in 5 seconds
console.log(getWithExpiry("token")); // Output: "abc123"
setTimeout(() => console.log(getWithExpiry("token")), 6000); // Output: null

```

**Explanation:**

**Set Expiry:**

- Store both the value and expiry time in local storage.

**Check Expiry:**

- On retrieval, compare the current time with the stored expiry time.

**Debounce and Throttle Function:**

**Task 46: Write a debounce function that delays the execution of a given function until after a specified time has passed since the last time it was called.**

**Solution:**

```

function debounce(fn, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}

// Example Usage
const logMessage = (message) => console.log(message);

const debouncedLog = debounce(logMessage, 1000);

debouncedLog("Hello"); // Only logs after 1 second if no new call occurs

```

**Explanation:**

**Concept:**

- Delays execution of fn until after delay milliseconds have passed since the last call.

#### **clearTimeout:**

- Resets the timer whenever the function is called again within the delay period.

**Task 47: Write a throttle function that ensures a given function is executed at most once every specified interval.**

#### **Solution:**

```
function throttle(fn, interval) {
  let lastExecuted = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastExecuted >= interval) {
      lastExecuted = now;
      fn(...args);
    }
  };
}

// Example Usage
const logScroll = () => console.log("Scroll event");

const throttledLog = throttle(logScroll, 1000);

window.addEventListener("scroll", throttledLog);
```

#### **Explanation:**

#### **Concept:**

- Ensures fn is called only once every interval milliseconds.

#### **Date.now:**

- Tracks the last execution time to control subsequent calls.

**Task 48: Throttle the execution of a function that logs the scroll position to the console, ensuring it runs at most once every 200ms.**

#### **Solution:**

```
function throttle(fn, interval) {
  let lastExecuted = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastExecuted >= interval) {
      lastExecuted = now;
      fn(...args);
    }
  };
}
```

```

    };
}

const logScrollPosition = () => {
  console.log(`Scroll position: ${window.scrollY}`);
};

const throttledScroll = throttle(logScrollPosition, 200);

window.addEventListener("scroll", throttledScroll);

```

**Explanation:**

**Throttling:**

- Ensures logScrollPosition executes at most once every 200ms, even if the scroll event fires more frequently.

**Task 49: Create a function that:**

- **Debounces a user typing in a search bar.**
- **Throttles the API call to prevent it from being made more than once every second.**

**Solution:**

```

function debounce(fn, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}

function throttle(fn, interval) {
  let lastExecuted = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastExecuted >= interval) {
      lastExecuted = now;
      fn(...args);
    }
  };
}

const fetchResults = (query) => {
  console.log(`Fetching results for: ${query}`);
};

const debouncedSearch = debounce((query) => throttledAPI(query), 300);
const throttledAPI = throttle(fetchResults, 1000);

```

```
// Simulate typing
debouncedSearch("JavaScript");
```

**Explanation:**

**Debouncing:**

- Waits until the user stops typing for 300ms before triggering the API call.

**Throttling:**

- Ensures the API call executes at most once every second.

## Task 50: Fetch a list of users from the JSONPlaceholder API and log their names to the console.

**Solution:**

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    data.forEach((user) => console.log(user.name));
  })
  .catch((error) => console.error("Error fetching data:", error));
```

**Explanation:**

1. **fetch:**
  - Makes a GET request to the API.
2. **Error Handling:**
  - Checks if the response status is OK (response.ok).
3. **Data Parsing:**
  - Converts the response to JSON format with response.json().

## Task 51: Send a new post object to the JSONPlaceholder API and log the server's response.

```
const newPost = {
  title: "My New Post",
  body: "This is the content of the post.",
  userId: 1,
};
```

**Solution:**

```
const newPost = {
```

```
title: "My New Post",
body: "This is the content of the post.",
userId: 1,
};

fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(newPost),
})
.then((response) => {
  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  }
  return response.json();
})
.then((data) => console.log("Server Response:", data))
.catch((error) => console.error("Error posting data:", error));
```

## Explanation

1. **Method:**
  - o Specifies the HTTP method (POST).
2. **Headers:**
  - o Defines the Content-Type as application/json.
3. **Body:**
  - o Sends the data as a JSON string using JSON.stringify.

## 1. Arrays & List Processing

### Pair Matching in Arrays (Basic Version)

**1. Problem:** Given an array of integers `nums` and a target integer `target`, return the indices of the two numbers that add up to the target.

**Example:**

```
Input: nums = [2, 7, 11, 15], target = 9
Output: [0, 1]
Explanation: nums[0] + nums[1] = 2 + 7 = 9
```

**Solution:**

```
function twoSum(nums, target) {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(complement)) {
      return [map.get(complement), i];
    }
    map.set(nums[i], i);
  }
  return [];
}

// Test
console.log(twoSum([2, 7, 11, 15], 9)); // Output: [0, 1]
```

**Explanation:**

1. **Objective:** Find two numbers in the array that add up to the target, and return their indices.
2. **Approach:** Use a hash map (Map) to store numbers as keys and their indices as values.
3. **Steps:**
  - o Iterate through the array with a for loop.
  - o For each number, calculate its complement (`target - nums[i]`).
  - o Check if the complement exists in the map:
    - If yes, return the indices of the complement (from the map) and the current number.
    - If no, store the current number and its index in the map.
4. **Why it Works:** The hash map allows O(1) lookup time for complements, ensuring an efficient solution.
5. **Time Complexity:** O(n) — Single pass through the array.
6. **Space Complexity:** O(n) — Space used by the hash map.

## Pair Matching in Arrays (Unique Pairs)

**2. Problem:** Find all unique pairs of numbers in the array that add up to the target. Ignore duplicate pairs.

**Example:**

Input: (nums = [1, 2, 3, 4, 3, 5]), (target = 6);

Output: [

[1, 5],

[2, 4],

[3, 3],

];

**Solution:**

```
function twoSumUniquePairs(nums, target) {  
    const seen = new Set();  
    const pairs = new Set();  
  
    nums.forEach((num) => {  
        const complement = target - num;  
        if (seen.has(complement)) {  
            const pair = [Math.min(num, complement), Math.max(num, complement)];  
            pairs.add(pair.toString());  
        }  
        seen.add(num);  
    });  
  
    return Array.from(pairs).map((pair) => pair.split(",").map(Number));  
}  
  
// Test  
console.log(twoSumUniquePairs([1, 2, 3, 4, 3, 5], 6)); // Output: [[1, 5], [2, 4], [3, 3]]
```

**Explanation:**

1. **Objective:** Find all unique pairs of numbers in the array that sum up to the target.
2. **Approach:** Use two sets:
  - o seen: To track numbers we've encountered.
  - o pairs: To store unique pairs as strings (to avoid duplicates).
3. **Steps:**
  - o For each number in the array:
    - Calculate its complement ( $\text{target} - \text{num}$ ).
    - If the complement exists in seen, create a pair with the smaller number first.
    - Convert the pair to a string and store it in pairs for uniqueness.

- Add the current number to the seen set.
  - Convert the pairs set back to an array of numbers and return it.
- Why it Works:** Sets ensure uniqueness, and Math.min/Math.max standardizes the order of pairs.
  - Time Complexity:** O(n) — Single pass through the array.
  - Space Complexity:** O(n) — Space used by the two sets.

### Pair Matching in Arrays (All Pairs of Indices)

#### 3. Problem: Return all pairs of indices whose numbers add up to the target.

##### Example:

```
Input: (nums = [1, 2, 3, 2]), (target = 4);
Output: [
  [0, 2],
  [1, 3],
];
```

##### Solution:

```
function twoSumAllPairs(nums, target) {
  const result = [];
  const map = new Map();

  nums.forEach((num, i) => {
    const complement = target - num;
    if (map.has(complement)) {
      map.get(complement).forEach((index) => result.push([index, i]));
    }
    if (!map.has(num)) {
      map.set(num, []);
    }
    map.get(num).push(i);
  });

  return result;
}

// Test
console.log(twoSumAllPairs([1, 2, 3, 2], 4)); // Output: [[0, 2], [1, 3]]
```

##### Explanation:

- Objective:** Find all index pairs (i, j) such that  $\text{nums}[i] + \text{nums}[j] = \text{target}$ .
- Approach:** Use a hash map (Map) to store each number's indices.
- Steps:**
  - For each number, calculate its complement ( $\text{target} - \text{num}$ ).

- If the complement exists in the map, add all pairs of indices [index, i] to the result.
  - Add the current index i to the list of indices for the current number in the map.
- Why it Works:** By storing indices for each number, we can efficiently find all pairs without nested loops.
  - Time Complexity:**  $O(n)$  — Single pass through the array with hash map operations.
  - Space Complexity:**  $O(n)$  — Space used by the hash map.

### Pair Matching in Arrays (Sorted Array)

**4. Problem: If the input array is sorted, return the indices of the two numbers that add up to the target.**

**Example:**

```
Input: (nums = [1, 2, 3, 4, 6]), (target = 6);
Output: [0, 3];
```

**Solution:**

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

  while (left < right) {
    const sum = nums[left] + nums[right];
    if (sum === target) return [left, right];
    if (sum < target) left++;
    else right--;
  }

  return [];
}

// Test
console.log(twoSumSorted([1, 2, 3, 4, 6], 6)); // Output: [0, 3]
```

**Explanation:**

- Objective:** Find two indices in a sorted array such that their sum equals the target.
- Approach:** Use the two-pointer technique:
  - Start with one pointer at the beginning (left) and the other at the end (right).
  - Calculate the sum of the numbers at these pointers.
- Steps:**
  - If the sum equals the target, return the indices.
  - If the sum is less than the target, move the left pointer right to increase the sum.

- If the sum is greater than the target, move the right pointer left to decrease the sum.
- Why it Works:** The sorted order ensures that adjusting the pointers efficiently narrows down the possibilities.
  - Time Complexity:** O(n) — Single traversal of the array.
  - Space Complexity:** O(1) — No additional space used.

### Pair Matching in Arrays (Closest to Target)

#### 5. Problem: Find the two numbers in the array whose sum is closest to the target.

##### Example:

Input: nums = [1, 2, 3, 4, 5], target = 10  
 Output: [4, 5]  
 Explanation: The sum  $4 + 5 = 9$  is the closest to 10.

##### Solution:

```
function twoSumClosest(nums, target) {
  nums.sort((a, b) => a - b); // Sort the array in ascending order
  let left = 0,
    right = nums.length - 1;
  let closestSum = Infinity;
  let result = [];

  while (left < right) {
    const sum = nums[left] + nums[right];
    if (Math.abs(target - sum) < Math.abs(target - closestSum)) {
      closestSum = sum;
      result = [nums[left], nums[right]];
    }
    if (sum < target) left++; // Increase the sum
    else right--; // Decrease the sum
  }

  return result;
}

// Test
console.log(twoSumClosest([1, 2, 3, 4, 5], 10)); // Output: [4, 5]
```

##### Explanation:

- Objective:** Find two numbers whose sum is closest to the target.
- Approach:**
  - Sort the array to efficiently navigate possible sums using two pointers.
  - Use two pointers (left at the start and right at the end) to calculate sums.
- Steps:**

- Calculate the current sum of  $\text{nums}[\text{left}]$  and  $\text{nums}[\text{right}]$ .
  - If the difference between the sum and the target is smaller than the difference for  $\text{closestSum}$ , update  $\text{closestSum}$  and the result.
  - Adjust pointers:
    - If the sum is smaller than the target, move left right to increase the sum.
    - If the sum is larger, move right left to decrease the sum.
4. **Why it Works:** Sorting ensures that we only need one pass with two pointers to find the closest sum.
  5. **Time Complexity:**  $O(n \log n)$  — Sorting takes  $O(n \log n)$ , and the two-pointer traversal is  $O(n)$ .
  6. **Space Complexity:**  $O(1)$  — Uses constant space apart from the result variable.

### Pair Matching in Arrays with Indices Difference Constraint

**6. Problem: Find two numbers that add up to the target, but the difference between their indices should not exceed k.**

**Example:**

```
Input: nums = [1, 2, 3, 4, 5], target = 6, k = 2
Output: [0, 2]
Explanation: nums[0] + nums[2] = 1 + 3 = 6, and |2 - 0| = 2 ≤ k.
```

**Solution:**

```
function twoSumWithConstraint(nums, target, k) {
  const map = new Map();

  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(complement) && i - map.get(complement) <= k) {
      return [map.get(complement), i];
    }
    map.set(nums[i], i);
  }

  return [];
}

// Test
console.log(twoSumWithConstraint([1, 2, 3, 4, 5], 6, 2)); // Output: [0, 2]
```

**Explanation:**

1. **Objective:** Find two numbers whose sum equals the target, with an additional constraint on their indices.
2. **Approach:**

- Use a hash map (Map) to store numbers and their indices as you iterate through the array.
  - Check if the complement exists in the map and if the indices difference meets the constraint.
3. **Steps:**
- For each number:
    - Calculate its complement ( $\text{target} - \text{nums}[i]$ ).
    - If the complement exists in the map and the index difference is  $\leq k$ , return the indices.
    - Otherwise, store the current number and its index in the map.
4. **Why it Works:** The hash map ensures efficient lookups for complements, and the condition on indices is checked during the iteration.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(n)$  — Hash map stores numbers and their indices.

### Pair Matching in Arrays in a Circular Array

7. **Problem:** Treat the array as circular. Find the indices of two numbers that add up to the target.

**Example:**

```
Input: nums = [3, 8, 12, 7], target = 10
Output: [0, 3]
Explanation: nums[0] + nums[3] = 3 + 7 = 10.
```

**Solution:**

```
function twoSumCircular(nums, target) {
  const n = nums.length;
  for (let i = 0; i < n; i++) {
    for (let j = 1; j < n; j++) {
      if (nums[i] + nums[(i + j) % n] === target) {
        return [i, (i + j) % n];
      }
    }
  }
  return [];
}

// Test
console.log(twoSumCircular([3, 8, 12, 7], 10)); // Output: [0, 3]
```

**Explanation:**

1. **Objective:** Find two numbers that sum up to the target in a circular array (last element connects back to the first).
2. **Approach:**
  - Use nested loops:

- The outer loop iterates through each number as the first number of the pair.
- The inner loop calculates sums with subsequent numbers, using  $(i + j) \% n$  to simulate circular behavior.

### 3. Steps:

- For each pair, calculate the sum of the current number and the number at the circular index.
- If the sum matches the target, return the indices.

4. **Why it Works:** The modulo operation handles circular behavior effectively.

5. **Time Complexity:**  $O(n^2)$  — Nested loops check all pairs.

6. **Space Complexity:**  $O(1)$  — No additional space used.

## Pair Matching in Arrays with Large Input

**8. Problem:** **Find two numbers in a large input array that add up to the target.**

### Example:

Input: `nums = [1, 3, 5, 7, 9, ... up to 1 million elements]`, target = 12  
 Output: `[1, 4]`

### Solution:

```
function twoSumLargeInput(nums, target) {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(complement)) {
      return [map.get(complement), i];
    }
    map.set(nums[i], i);
  }
  return [];
}

// Test
console.log(twoSumLargeInput([1, 3, 5, 7, 9], 12)); // Output: [1, 4]
```

### Explanation:

1. **Objective:** Efficiently find two numbers in a very large array that add up to the target.

### 2. Approach:

- Use a hash map to track numbers and their indices as you iterate.
- Check if the complement of the current number exists in the map.

### 3. Steps:

- For each number:
  - Calculate the complement ( $\text{target} - \text{nums}[i]$ ).
  - If the complement exists in the map, return the indices.

- Otherwise, store the current number and its index in the map.
4. **Why it Works:** The hash map allows O(1) lookups, ensuring the solution is efficient for large arrays.
  5. **Time Complexity:** O(n) — Single traversal.
  6. **Space Complexity:** O(n) — Hash map grows linearly with input size.

### Pair Matching in Arrays (Multidimensional Array)

**9. Problem:** Given a 2D array of integers, find two numbers from the array that add up to the target. Return their coordinates as [row1, col1], [row2, col2].

**Example:**

```
Input: nums = [[1, 2], [3, 4], [5, 6]], target = 9
Output: [[1, 1], [2, 0]]
Explanation: nums[1][1] + nums[2][0] = 4 + 5 = 9.
```

**Solution:**

```
function twoSum2D(nums, target) {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    for (let j = 0; j < nums[i].length; j++) {
      const complement = target - nums[i][j];
      if (map.has(complement)) {
        return [map.get(complement), [i, j]];
      }
      map.set(nums[i][j], [i, j]);
    }
  }
  return [];
}

// Test
console.log(
  twoSum2D(
    [
      [1, 2],
      [3, 4],
      [5, 6],
    ],
    9
  )
); // Output: [[1, 1], [2, 0]]
```

**Explanation:**

1. **Objective:** Find two numbers in a 2D array that sum up to the target and return their coordinates.

## 2. Approach:

- o Use a hash map (Map) to store each number as a key and its coordinates [row, col] as the value.
- o Iterate through the 2D array using nested loops.

## 3. Steps:

- o For each element, calculate the complement ( $\text{target} - \text{nums}[i][j]$ ).
- o Check if the complement exists in the map:
  - If it does, return the coordinates of the complement and the current element.
  - If it doesn't, store the current element and its coordinates in the map.

4. Why it Works: The hash map allows  $O(1)$  lookups for complements, making the solution efficient even for larger 2D arrays.

5. Time Complexity:  $O(n * m)$  — Where  $n$  is the number of rows and  $m$  is the number of columns.

6. Space Complexity:  $O(n * m)$  — The hash map can store up to  $n * m$  elements.

## Pair Matching in Arrays (Count All Pairs)

**10. Problem:** Find the count of all unique pairs of numbers that add up to the target. A pair should only be counted once, even if the same numbers appear multiple times.

### Example:

Input:  $\text{nums} = [1, 1, 2, 3, 4, 5]$ ,  $\text{target} = 6$

Output: 2

Explanation: [1, 5] and [2, 4] are the two pairs that sum to 6.

### Solution:

```
function twoSumCount(nums, target) {  
    const map = new Map();  
    let count = 0;  
  
    nums.forEach((num) => {  
        const complement = target - num;  
        if (map.has(complement) && map.get(complement) > 0) {  
            count++;  
            map.set(complement, map.get(complement) - 1); // Decrement count of complement  
        } else {  
            map.set(num, (map.get(num) || 0) + 1); // Increment count of current number  
        }  
    });  
  
    return count;  
}  
  
// Test  
console.log(twoSumCount([1, 1, 2, 3, 4, 5], 6)); // Output: 2
```

## **Explanation:**

1. **Objective:** Count the number of unique pairs of numbers that add up to the target.
2. **Approach:**
  - o Use a hash map to track the frequency of each number as you iterate through the array.
3. **Steps:**
  - o For each number, calculate its complement ( $\text{target} - \text{num}$ ).
  - o If the complement exists in the map and has a positive count:
    - Increment the count of valid pairs.
    - Decrement the frequency of the complement to avoid counting it again.
  - o If the complement doesn't exist or has been used, increment the frequency of the current number in the map.
4. **Why it Works:** The hash map ensures efficient lookups and avoids counting duplicate pairs.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(n)$  — Space used by the hash map.

## **Stock Price Optimization:**

### **Stock Price Optimization (Max Profit with One Transaction)**

**11. Problem:** You are given an array prices where prices[i] is the price of a given stock on day i. Return the maximum profit you can achieve from one buy and one sell. If you cannot achieve any profit, return 0.

#### **Example:**

Input: prices = [7, 1, 5, 3, 6, 4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6 - 1 = 5.

#### **Solution:**

```
function maxProfit(prices) {  
    let minPrice = Infinity; // Track the lowest price  
    let maxProfit = 0; // Track the maximum profit  
  
    for (let price of prices) {  
        minPrice = Math.min(minPrice, price); // Update the minimum price  
        maxProfit = Math.max(maxProfit, price - minPrice); // Calculate the max profit  
    }  
  
    return maxProfit;  
}  
  
// Test
```

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

### Explanation:

1. **Objective:** Maximize profit by buying and selling once.
2. **Approach:** Keep track of:
  - o The minimum price (minPrice) seen so far.
  - o The maximum profit (maxProfit) possible at any given day.
3. **Steps:**
  - o Loop through the prices array.
  - o For each price:
    - Update minPrice with the lower of the current price or the previous minPrice.
    - Calculate the profit if you sold at the current price (price - minPrice) and update maxProfit if this profit is higher.
4. **Why it Works:** By maintaining minPrice dynamically, you ensure you always calculate the profit from the best buying opportunity up to the current day.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Only two variables (minPrice and maxProfit) are used.

## Stock Price Optimization II (Multiple Transactions Allowed)

**12. Problem: You can complete as many transactions as you like (buy and sell multiple times). However, you must sell the stock before buying again.**

### Example:

Input: prices = [7, 1, 5, 3, 6, 4]

Output: 7

Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit =  $5 - 1 = 4$ .

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit =  $6 - 3 = 3$ .

### Solution:

```
function maxProfitMultiple(prices) {  
    let profit = 0;  
  
    for (let i = 1; i < prices.length; i++) {  
        if (prices[i] > prices[i - 1]) {  
            profit += prices[i] - prices[i - 1]; // Add profit from consecutive days  
        }  
    }  
  
    return profit;  
}  
  
// Test  
console.log(maxProfitMultiple([7, 1, 5, 3, 6, 4])); // Output: 7
```

### **Explanation:**

1. **Objective:** Maximize profit by making multiple buy-sell transactions.
2. **Approach:** Add up all the "upward" movements in stock prices:
  - o If  $\text{prices}[i] > \text{prices}[i-1]$ , there is a profit to be made, so add the difference to the total profit.
3. **Steps:**
  - o Start iterating from the second day.
  - o For each day:
    - If the stock price is higher than the previous day, calculate the profit ( $\text{prices}[i] - \text{prices}[i-1]$ ) and add it to the total.
  - o Skip days where the price decreases or stays the same.
4. **Why it Works:** Adding only positive differences ensures that you always "buy low and sell high" whenever the opportunity exists.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Only a single profit variable is used.

### **Stock Price Optimization with Transaction Fee**

**13. Problem: Each time you sell a stock, you pay a transaction fee. Calculate the maximum profit you can achieve.**

#### **Example:**

Input:  $\text{prices} = [1, 3, 2, 8, 4, 9]$ ,  $\text{fee} = 2$

Output: 8

Explanation: Buy on day 1 (price = 1) and sell on day 4 (price = 8), profit =  $8 - 1 - 2 = 5$ . Then buy on day 5 (price = 4) and sell on day 6 (price = 9), profit =  $9 - 4 - 2 = 3$ .

#### **Solution:**

```
function maxProfitWithFee(prices, fee) {  
    let hold = -Infinity; // Maximum profit holding a stock  
    let cash = 0; // Maximum profit not holding a stock  
  
    for (let price of prices) {  
        const prevCash = cash;  
        cash = Math.max(cash, hold + price - fee); // Max profit if we sell today  
        hold = Math.max(hold, prevCash - price); // Max profit if we buy today  
    }  
  
    return cash;  
}  
  
// Test  
console.log(maxProfitWithFee([1, 3, 2, 8, 4, 9], 2)); // Output: 8
```

### **Explanation:**

1. **Objective:** Maximize profit with multiple transactions, accounting for a fixed transaction fee for each sale.
2. **Approach:** Use two variables:
  - o hold: Maximum profit if you are holding a stock.
  - o cash: Maximum profit if you are not holding a stock.
3. **Steps:**
  - o For each day:
    - Update cash as the maximum of:
      - Keeping the previous cash.
      - Selling the stock held (hold + price - fee).
    - Update hold as the maximum of:
      - Keeping the previous hold.
      - Buying a new stock (cash - price).
4. **Why it Works:** This dynamic programming approach ensures that the state transitions accurately reflect the possible actions for each day.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Only two variables (hold and cash) are used.

### Stock Price Optimization with Cooldown

**14. Problem:** You can complete as many transactions as you like, but after you sell a stock, you cannot buy on the next day (cooldown period).

**Example:**

Input: `prices = [1, 2, 3, 0, 2]`

Output: 3

Explanation: Buy on day 1 (price = 1), sell on day 3 (price = 3), cooldown on day 4, and buy on day 5 (price = 0), sell on day 5 (price = 2).

**Solution:**

```
function maxProfitWithCooldown(prices) {
  let sell = 0; // Maximum profit after selling a stock
  let hold = -Infinity; // Maximum profit after buying a stock
  let cooldown = 0; // Maximum profit during cooldown

  for (let price of prices) {
    const prevSell = sell;
    sell = hold + price; // Sell the stock held
    hold = Math.max(hold, cooldown - price); // Buy a stock after cooldown
    cooldown = Math.max(cooldown, prevSell); // Transition to cooldown
  }

  return Math.max(sell, cooldown); // Return the maximum profit
}

// Test
```

```
console.log(maxProfitWithCooldown([1, 2, 3, 0, 2])); // Output: 3
```

### Explanation:

1. **Objective:** Maximize profit while adhering to a cooldown period after selling.
2. **Approach:**
  - o Use three states:
    - sell: The maximum profit achievable after selling a stock.
    - hold: The maximum profit achievable while holding a stock.
    - cooldown: The maximum profit achievable during the cooldown period.
  - o Transition between states based on daily actions.
3. **Steps:**
  - o For each day's price:
    - Update sell to reflect the profit after selling the stock held.
    - Update hold to reflect the profit after buying a stock, either from cooldown or continuing to hold.
    - Update cooldown to reflect the profit from the previous sell or staying in cooldown.
  - o At the end, return the maximum of sell and cooldown, since holding a stock on the last day does not yield profit.
4. **Why it Works:** The transitions between states ensure all possible scenarios (buy, sell, cooldown) are covered.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Only three variables (sell, hold, cooldown) are used.

## Stock Price Optimization III (At Most Two Transactions)

**15. Problem: You are allowed to complete at most two transactions. Calculate the maximum profit.**

### Example:

Input: prices = [3, 3, 5, 0, 0, 3, 1, 4]

Output: 6

Explanation: Buy on day 4 (price = 0), sell on day 6 (price = 3), then buy on day 7 (price = 1), and sell on day 8 (price = 4).

### Solution:

```
function maxProfitTwoTransactions(prices) {  
    let buy1 = -Infinity,  
        sell1 = 0;  
    let buy2 = -Infinity,  
        sell2 = 0;  
  
    for (let price of prices) {
```

```

        sell2 = Math.max(sell2, buy2 + price); // Sell after the second buy
        buy2 = Math.max(buy2, sell1 - price); // Buy after the first sell
        sell1 = Math.max(sell1, buy1 + price); // Sell after the first buy
        buy1 = Math.max(buy1, -price); // First buy
    }

    return sell2; // Maximum profit after at most two transactions
}

// Test
console.log(maxProfitTwoTransactions([3, 3, 5, 0, 0, 3, 1, 4])); // Output: 6

```

### **Explanation:**

1. **Objective:** Maximize profit with at most two buy-sell transactions.
2. **Approach:**
  - o Use four variables:
    - buy1 and sell1 for the first transaction.
    - buy2 and sell2 for the second transaction.
  - o Update these variables based on whether to buy or sell for each transaction.
3. **Steps:**
  - o For each day's price:
    - Update sell2 to maximize profit after the second sell.
    - Update buy2 to maximize profit after the second buy.
    - Update sell1 to maximize profit after the first sell.
    - Update buy1 to maximize profit after the first buy.
  - o Return sell2 as it represents the maximum profit after two transactions.
4. **Why it Works:** Each variable ensures the profit is maximized step by step for each transaction.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Only four variables are used.

### **Stock Price Optimization IV (At Most K Transactions)**

**16. Problem: You are allowed to complete at most k transactions. Calculate the maximum profit.**

#### **Example:**

Input: prices = [3, 2, 6, 5, 0, 3], k = 2  
Output: 7  
Explanation: Buy on day 2 (price = 2), sell on day 3 (price = 6), then buy on day 5 (price = 0), and sell on day 6 (price = 3).

#### **Solution:**

```

function maxProfitKTransactions(prices, k) {
    if (prices.length === 0) return 0;
}

```

```

const dp = Array.from({ length: k + 1 }, () => Array(prices.length).fill(0));

for (let t = 1; t <= k; t++) {
    let maxDiff = -prices[0];
    for (let d = 1; d < prices.length; d++) {
        dp[t][d] = Math.max(dp[t][d - 1], prices[d] + maxDiff);
        maxDiff = Math.max(maxDiff, dp[t - 1][d] - prices[d]);
    }
}

return dp[k][prices.length - 1];
}

// Test
console.log(maxProfitKTransactions([3, 2, 6, 5, 0, 3], 2)); // Output: 7

```

### **Explanation:**

1. **Objective:** Maximize profit with at most k buy-sell transactions.
2. **Approach:**
  - o Use dynamic programming ( $dp[t][d]$ ) where:
    - t is the number of transactions allowed.
    - d is the day.
    - $dp[t][d]$  represents the maximum profit achievable with t transactions by day d.
  - o Use maxDiff to optimize the inner loop.
3. **Steps:**
  - o Initialize a 2D dp array with dimensions  $(k + 1) \times \text{prices.length}$ .
  - o For each transaction t:
    - Track the maximum difference (maxDiff) between profits and prices.
    - For each day d:
      - Update  $dp[t][d]$  as the maximum of:
        - Skipping the transaction ( $dp[t][d - 1]$ ).
        - Selling the stock bought on any previous day ( $\text{prices}[d] + \text{maxDiff}$ ).
      - Update maxDiff.
    - o Return the maximum profit for k transactions on the last day.
4. **Why it Works:** Dynamic programming ensures all possible transactions and profits are considered efficiently.
5. **Time Complexity:**  $O(k * n)$  — Outer loop for transactions, inner loop for days.
6. **Space Complexity:**  $O(k * n)$  — Space used by the dp array.

### **Stock Price Optimization with Infinite Transactions and Cooldown**

**17. Problem:** You can complete as many transactions as you like, but after you sell a stock, you cannot buy on the next day (cooldown period).

### **Example:**

Input: prices = [1, 2, 3, 0, 2]

Output: 3

Explanation: Buy on day 1 (price = 1), sell on day 3 (price = 3), cooldown on day 4, then buy on day 5 (price = 0), sell on day 5 (price = 2).

### Solution:

```
function maxProfitInfiniteWithCooldown(prices) {  
    let sell = 0,  
        hold = -Infinity,  
        cooldown = 0;  
  
    for (let price of prices) {  
        const prevSell = sell;  
        sell = hold + price; // Sell the stock held  
        hold = Math.max(hold, cooldown - price); // Buy the stock after cooldown  
        cooldown = Math.max(cooldown, prevSell); // Transition to cooldown  
    }  
  
    return Math.max(sell, cooldown);  
}  
  
// Test  
console.log(maxProfitInfiniteWithCooldown([1, 2, 3, 0, 2])); // Output: 3
```

### Explanation:

1. **Objective:** Maximize profit while adhering to the cooldown restriction after each sale.
2. **Approach:** Use three states:
  - o sell: The maximum profit after selling a stock.
  - o hold: The maximum profit while holding a stock.
  - o cooldown: The maximum profit during a cooldown period.
3. **Steps:**
  - o Iterate over each price.
  - o Update the states dynamically:
    - sell is updated as the profit from selling a stock held earlier.
    - hold is updated as the maximum of continuing to hold a stock or buying a new one after cooldown.
    - cooldown is updated as the maximum profit from remaining in cooldown or transitioning from a sale.
4. **Why it Works:** The transitions account for all possible actions (sell, hold, cooldown) efficiently.
5. **Time Complexity:** O(n) — Single traversal of the prices array.
6. **Space Complexity:** O(1) — Uses only three variables (sell, hold, cooldown).

### Stock Price Optimization with Transaction Fee and Cooldown

**18. Problem:** Each time you sell a stock, you pay a transaction fee. After selling, you must also wait one day before buying again (cooldown).

**Example:**

Input: prices = [1, 3, 2, 8, 4, 9], fee = 2

Output: 8

Explanation: Buy on day 1 (price = 1), sell on day 4 (price = 8), cooldown on day 5, then buy on day 6 (price = 4), sell on day 6 (price = 9).

**Solution:**

```
function maxProfitFeeCooldown(prices, fee) {  
    let sell = 0,  
        hold = -Infinity,  
        cooldown = 0;  
  
    for (let price of prices) {  
        const prevSell = sell;  
        sell = Math.max(sell, hold + price - fee); // Sell and pay fee  
        hold = Math.max(hold, cooldown - price); // Buy after cooldown  
        cooldown = prevSell; // Transition to cooldown  
    }  
  
    return sell;  
}  
  
// Test  
console.log(maxProfitFeeCooldown([1, 3, 2, 8, 4, 9], 2)); // Output: 8
```

**Explanation:**

1. **Objective:** Maximize profit with transaction fees and cooldown periods after each sale.
2. **Approach:** Similar to the infinite transactions problem with cooldown, but add a transaction fee when selling.
3. **Steps:**
  - o Iterate through prices:
    - Update sell by considering the profit after selling a stock and subtracting the fee.
    - Update hold by considering the maximum profit after buying a stock, accounting for the cooldown period.
    - Update cooldown to reflect the profit after a sale.
4. **Why it Works:** This approach dynamically transitions between the states (sell, hold, cooldown), while also accounting for the fee.
5. **Time Complexity:** O(n) — Single traversal of the prices array.
6. **Space Complexity:** O(1) — Uses only three variables.

## Stock Price Optimization (Max Profit with Three Transactions)

**19. Problem:** You are allowed to complete at most three transactions. Calculate the maximum profit.

**Example:**

Input: prices = [3, 3, 5, 0, 0, 3, 1, 4]

Output: 6

Explanation: Buy on day 4 (price = 0), sell on day 6 (price = 3), buy on day 7 (price = 1), and sell on day 8 (price = 4).

**Solution:**

```
function maxProfitThreeTransactions(prices) {  
    let buy1 = -Infinity,  
        sell1 = 0;  
    let buy2 = -Infinity,  
        sell2 = 0;  
    let buy3 = -Infinity,  
        sell3 = 0;  
  
    for (let price of prices) {  
        sell3 = Math.max(sell3, buy3 + price);  
        buy3 = Math.max(buy3, sell2 - price);  
        sell2 = Math.max(sell2, buy2 + price);  
        buy2 = Math.max(buy2, sell1 - price);  
        sell1 = Math.max(sell1, buy1 + price);  
        buy1 = Math.max(buy1, -price);  
    }  
  
    return sell3;  
}  
  
// Test  
console.log(maxProfitThreeTransactions([3, 3, 5, 0, 0, 3, 1, 4])); // Output: 6
```

**Explanation:**

1. **Objective:** Maximize profit with up to three buy-sell transactions.
2. **Approach:**
  - o Use six variables:
    - buy1, sell1: First transaction.
    - buy2, sell2: Second transaction.
    - buy3, sell3: Third transaction.
  - o Update these variables dynamically for each transaction.
3. **Steps:**

- For each day's price:
    - Update sell3 with the maximum profit after the third sell.
    - Update buy3 with the maximum profit after the third buy.
    - Similarly, update sell2, buy2, sell1, and buy1.
4. **Why it Works:** This approach iteratively builds up the maximum profit for three transactions in a dynamic and efficient manner.
  5. **Time Complexity:** O(n) — Single traversal of the prices array.
  6. **Space Complexity:** O(1) — Uses six variables for the states.

### **Stock Price Optimization (Infinite Transactions with K Cooldown Days)**

**20. Problem: You can complete as many transactions as you like, but after selling, you must wait k days before buying again.**

**Example:**

Input: (prices = [1, 2, 3, 0, 2]), (k = 2);  
Output: 3;

Solution:

```
function maxProfitWithKCooldown(prices, k) {
  const n = prices.length;
  if (n === 0) return 0;

  const dp = Array.from({ length: n }, () => Array(2).fill(0));
  dp[0][1] = -prices[0]; // Initial state: bought on day 0

  for (let i = 1; i < n; i++) {
    dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
    dp[i][1] = Math.max(
      dp[i - 1][1],
      (i >= k + 1 ? dp[i - k - 1][0] : 0) - prices[i]
    );
  }

  return dp[n - 1][0];
}

// Test
console.log(maxProfitWithKCooldown([1, 2, 3, 0, 2], 2)); // Output: 3
```

**Explanation:**

1. **Objective:** Maximize profit with infinite transactions and a cooldown period of k days.
2. **Approach:**

- Use dynamic programming (dp array) to track:
    - $dp[i][0]$ : Maximum profit on day  $i$  without holding a stock.
    - $dp[i][1]$ : Maximum profit on day  $i$  while holding a stock.
  - Update states based on possible actions (buy, sell, or cooldown).
3. **Steps:**
- Iterate through the prices array, updating  $dp[i][0]$  and  $dp[i][1]$  based on the previous states.
4. **Time Complexity:**  $O(n)$  — Single traversal.  
 5. **Space Complexity:**  $O(n)$  — Space for the dp array.

## Optimizing Array Search

### Optimizing Array Search (No Duplicates)

**21. Problem:** A sorted array has been rotated at some pivot unknown to you beforehand. Find the minimum element in the array. Assume there are no duplicate elements.

**Example:**

```
Input: nums = [3, 4, 5, 1, 2]
Output: 1
Explanation: The minimum value in the array is 1.
```

**Solution:**

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

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

    if (nums[mid] > nums[right]) {
      left = mid + 1; // The minimum is in the right half
    } else {
      right = mid; // The minimum is in the left half or is mid
    }
  }

  return nums[left];
}

// Test
console.log(findMin([3, 4, 5, 1, 2])); // Output: 1
```

**Explanation:**

1. **Objective:** Find the minimum element in a rotated sorted array.
2. **Approach:** Use a modified binary search to efficiently find the minimum:
  - o Compare the middle element (`nums[mid]`) with the last element (`nums[right]`) to decide which half of the array to search.
3. **Steps:**
  - o If `nums[mid] > nums[right]`, the minimum must be in the right half.
  - o Otherwise, the minimum is in the left half or could be the middle element itself.
  - o Adjust the left and right pointers accordingly and repeat.
4. **Why it Works:** Rotated sorted arrays have a property that the smallest element divides the array into two sorted halves. Binary search exploits this property to reduce the search space.
5. **Time Complexity:**  $O(\log n)$  — Binary search reduces the search space by half in each iteration.
6. **Space Complexity:**  $O(1)$  — No additional space is used.

## Optimizing Array Search II (With Duplicates)

**22. Problem: A sorted array has been rotated at some pivot unknown to you beforehand. The array may contain duplicates. Find the minimum element in the array.**

**Example:**

```
Input: nums = [2, 2, 2, 0, 1]
Output: 0
Explanation: The minimum value in the array is 0.
```

**Solution:**

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

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

    if (nums[mid] > nums[right]) {
      left = mid + 1; // The minimum is in the right half
    } else if (nums[mid] < nums[right]) {
      right = mid; // The minimum is in the left half or is mid
    } else {
      right--; // Skip duplicate by decrementing the right pointer
    }
  }

  return nums[left];
}

// Test
```

```
console.log(findMinWithDuplicates([2, 2, 2, 0, 1])); // Output: 0
```

### Explanation:

1. **Objective:** Handle duplicate elements while finding the minimum in a rotated sorted array.
2. **Approach:** Use a binary search approach similar to the no-duplicate case, but add an additional step for handling duplicates.
3. **Steps:**
  - o If  $\text{nums}[\text{mid}] > \text{nums}[\text{right}]$ , the minimum is in the right half.
  - o If  $\text{nums}[\text{mid}] < \text{nums}[\text{right}]$ , the minimum is in the left half or is the middle element.
  - o If  $\text{nums}[\text{mid}] === \text{nums}[\text{right}]$ , decrement the right pointer to skip the duplicate.
4. **Why it Works:** By skipping duplicates, the search space is still reduced in each step while maintaining correctness.
5. **Time Complexity:**  $O(n)$  in the worst case (when all elements are duplicates), but  $O(\log n)$  on average.
6. **Space Complexity:**  $O(1)$  — No additional space is used.

### Optimizing Array Search (Using Linear Search)

#### 23. Problem: Write a solution using linear search to find the minimum element in a rotated sorted array.

##### Example:

```
Input: nums = [4, 5, 6, 7, 0, 1, 2]
Output: 0
Explanation: The minimum value in the array is 0.
```

##### Solution:

```
function findMinLinear(nums) {
  let min = nums[0];

  for (let i = 1; i < nums.length; i++) {
    if (nums[i] < min) {
      min = nums[i];
    }
  }

  return min;
}

// Test
```

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

### Explanation:

1. **Objective:** Find the minimum element in a rotated sorted array using a straightforward linear search.
2. **Approach:** Iterate through the array while keeping track of the smallest element.
3. **Steps:**
  - o Initialize min with the first element.
  - o Loop through the array:
    - If the current element is smaller than min, update min.
  - o Return min after the loop.
4. **Why it Works:** By checking every element, this approach guarantees finding the minimum in all cases.
5. **Time Complexity:**  $O(n)$  — Requires checking every element in the array.
6. **Space Complexity:**  $O(1)$  — Only a single variable (min) is used.

### Optimizing Array Search (Recursive Approach)

#### 24. Problem: **Implement a recursive solution to find the minimum element in a rotated sorted array.**

### Example:

```
Input: nums = [4, 5, 6, 7, 0, 1, 2]
Output: 0
Explanation: The minimum value in the array is 0.
```

### Solution:

```
function findMinRecursive(nums, left = 0, right = nums.length - 1) {
  if (left === right) return nums[left];

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

  if (nums[mid] > nums[right]) {
    return findMinRecursive(nums, mid + 1, right); // Search the right half
  } else {
    return findMinRecursive(nums, left, mid); // Search the left half
  }
}

// Test
console.log(findMinRecursive([4, 5, 6, 7, 0, 1, 2])); // Output: 0
```

### Explanation:

1. **Objective:** Use recursion to implement a binary search for finding the minimum element.
2. **Approach:** Split the array into halves based on the relationship between `nums[mid]` and `nums[right]`.
3. **Steps:**
  - o If left equals right, return `nums[left]` as it is the minimum.
  - o Calculate the middle index mid.
  - o Compare `nums[mid]` with `nums[right]`:
    - If `nums[mid] > nums[right]`, search the right half (mid + 1 to right).
    - Otherwise, search the left half (left to mid).
4. **Why it Works:** Recursion narrows the search space using the same logic as iterative binary search.
5. **Time Complexity:**  $O(\log n)$  — Each recursive step halves the array.
6. **Space Complexity:**  $O(\log n)$  — Recursive stack depth proportional to the array size.

### Optimizing Array Search (Find Pivot and Determine Minimum)

**25. Problem: Find the minimum element in a rotated sorted array by first locating the pivot point (the index where the array rotation occurs).**

**Example:**

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`

Output: 0

Explanation: The pivot is at index 4, and the minimum value is `nums[4] = 0`.

**Solution:**

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

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

    if (nums[mid] > nums[right]) {
      left = mid + 1; // Pivot is in the right half
    } else {
      right = mid; // Pivot is in the left half or is mid
    }
  }

  return nums[left]; // Left will point to the pivot (minimum element)
}

// Test
```

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

### Explanation:

1. **Objective:** Locate the pivot point to determine the minimum element.
2. **Approach:** Similar to binary search but focuses on finding the point of rotation:
  - o If  $\text{nums}[\text{mid}] > \text{nums}[\text{right}]$ , the pivot lies in the right half.
  - o Otherwise, the pivot is in the left half or at mid.
3. **Steps:**
  - o Initialize left and right pointers.
  - o Narrow the search space using binary search rules.
  - o Once left equals right, the pivot (minimum element) is found.
4. **Why it Works:** The pivot divides the rotated array into two sorted halves. Binary search exploits this structure.
5. **Time Complexity:**  $O(\log n)$  — Halves the search space at each step.
6. **Space Complexity:**  $O(1)$  — No additional memory is used.

### Optimizing Array Search (Return Both Pivot and Minimum)

#### 26. Problem: Return both the pivot index and the minimum element in the rotated sorted array.

##### Example:

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`

Output: { pivot: 4, min: 0 }

Explanation: The pivot is at index 4, where the minimum value is 0.

##### Solution:

```
function findPivotAndMin(nums) {  
    let left = 0,  
        right = nums.length - 1;  
  
    while (left < right) {  
        const mid = Math.floor((left + right) / 2);  
  
        if (nums[mid] > nums[right]) {  
            left = mid + 1; // Pivot is in the right half  
        } else {  
            right = mid; // Pivot is in the left half or is mid  
        }  
    }  
  
    return { pivot: left, min: nums[left] };  
}
```

```
// Test
console.log(findPivotAndMin([4, 5, 6, 7, 0, 1, 2])); // Output: { pivot: 4, min: 0 }
```

### Explanation:

1. **Objective:** Identify both the pivot index and the minimum value.
2. **Approach:** Follow the same binary search logic as finding the pivot:
  - o left eventually points to the pivot, where the minimum element resides.
3. **Steps:**
  - o Perform binary search to locate the pivot.
  - o Return the pivot index (left) and the corresponding minimum value (nums[left]).
4. **Why it Works:** Binary search efficiently narrows down the search space to the pivot index.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — Uses no additional memory.

## Optimizing Array Search (Handle Very Large Arrays)

### 27. Problem: Optimize the solution to handle very large rotated sorted arrays efficiently.

#### Example:

Input: nums = [4, 5, 6, 7, ..., 0, 1, 2]

Output: 0

Explanation: The array is large, but the logic for finding the minimum remains unchanged.

#### Solution:

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

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

    if (nums[mid] > nums[right]) {
      left = mid + 1; // Pivot is in the right half
    } else {
      right = mid; // Pivot is in the left half or is mid
    }
  }

  return nums[left];
}
```

```
// Test
console.log(findMinLargeArray([4, 5, 6, 7, ...Array(1e6).fill(0), 1, 2])); // Output: 0
```

### Explanation:

1. **Objective:** Ensure that the solution remains efficient for very large arrays.
2. **Approach:** Use the same binary search technique:
  - o Avoid unnecessary iterations and focus only on narrowing the search space.
3. **Steps:**
  - o Perform binary search to locate the pivot where the minimum resides.
  - o Return the minimum value (nums[left]).
4. **Why it Works:** Binary search ensures the algorithm scales well with large input sizes.
5. **Time Complexity:**  $O(\log n)$  — Efficient even for large arrays.
6. **Space Complexity:**  $O(1)$  — Constant memory usage.

### Optimizing Array Search (Edge Cases)

**28. Problem: Account for edge cases in the array, such as when it is not rotated or contains a single element.**

#### Examples:

1. Input: nums = [1, 2, 3, 4, 5]  
Output: 1
2. Input: nums = [1]  
Output: 1
3. Input: nums = [2, 1]  
Output: 1

#### Solution:

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

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

    if (nums[mid] > nums[right]) {
      left = mid + 1;
    } else {
      right = mid;
    }
  }

  return nums[left];
}
```

```
// Test Cases
console.log(findMinEdgeCases([1, 2, 3, 4, 5])); // Output: 1
console.log(findMinEdgeCases([1])); // Output: 1
console.log(findMinEdgeCases([2, 1])); // Output: 1
```

### Explanation:

1. **Objective:** Handle special cases where the array is already sorted, has only one element, or contains just two elements.
2. **Approach:** Use binary search but ensure it works for edge cases:
  - o If the array is already sorted, binary search naturally returns the first element.
  - o If the array has one or two elements, the logic remains the same.
3. **Steps:**
  - o Perform binary search as usual, reducing the search space at each step.
  - o Return the element at the left pointer when the search ends.
4. **Why it Works:** Binary search is robust enough to handle these scenarios without additional checks.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No additional memory used.

### Optimizing Array Search (Duplicate Pivot Case)

**29. Problem:** Handle cases where the pivot duplicates the smallest value, and the array still needs to be sorted.

### Example:

```
Input: nums = [4, 5, 6, 0, 0, 1, 2]
Output: 0
Explanation: The minimum value is duplicated in the array.
```

### Solution:

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

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

        if (nums[mid] > nums[right]) {
            left = mid + 1; // Pivot is in the right half
        } else if (nums[mid] < nums[right]) {
            right = mid; // Pivot is in the left half or is mid
        } else {
            right--; // Skip duplicates
        }
    }

    return nums[left];
}
```

```

    }

    return nums[left];
}

// Test
console.log(findMinWithDuplicatePivot([4, 5, 6, 0, 0, 1, 2])); // Output: 0

```

### Explanation:

1. **Objective:** Locate the minimum even if the array contains duplicates, including repeated pivot values.
2. **Approach:** Use binary search logic and skip duplicates by reducing the search space.
3. **Steps:**
  - o Check if `nums[mid]` equals `nums[right]`, and decrement right to avoid duplicates.
  - o Narrow down the search space:
    - If `nums[mid] > nums[right]`, search the right half.
    - If `nums[mid] < nums[right]`, search the left half or at mid.
4. **Why it Works:** By ignoring duplicates during comparison, the algorithm ensures correctness while maintaining efficiency.
5. **Time Complexity:**  $O(n)$  in the worst case (when all elements are duplicates), but  $O(\log n)$  on average.
6. **Space Complexity:**  $O(1)$  — No extra memory used.

### Optimizing Array Search (Single Rotation Check)

**30. Problem:** Given a rotated sorted array, determine if there is exactly one rotation and find the minimum.

#### Example:

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`  
Output: 0  
Explanation: The array is rotated once, and the minimum value is 0.

#### Solution:

```

function findMinSingleRotation(nums) {
  if (nums[0] < nums[nums.length - 1]) {
    return nums[0]; // Already sorted, no rotation
  }

  let left = 0,
    right = nums.length - 1;

  while (left < right) {

```

```

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

if (nums[mid] > nums[right]) {
    left = mid + 1;
} else {
    right = mid;
}
}

return nums[left];
}

// Test
console.log(findMinSingleRotation([4, 5, 6, 7, 0, 1, 2])); // Output: 0
console.log(findMinSingleRotation([1, 2, 3, 4, 5])); // Output: 1

```

### **Explanation:**

1. **Objective:** Verify if the array is rotated once and find the minimum.
2. **Approach:** First check if the array is already sorted (no rotation):
  - o If  $\text{nums}[0] < \text{nums}[\text{nums.length} - 1]$ , return  $\text{nums}[0]$ .
  - o Otherwise, perform a binary search for the minimum.
3. **Steps:**
  - o Use the same logic as finding the pivot to locate the minimum.
  - o Return the minimum element when found.
4. **Why it Works:** A single rotation divides the array into two sorted parts, which binary search can handle efficiently.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No extra memory used.

## **Maximum Subarray**

### **Maximum Subarray (Kadane's Algorithm)**

**31. Problem:** **Find the contiguous subarray (containing at least one number) that has the largest sum, and return its sum.**

### **Example:**

Input:  $\text{nums} = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$   
 Output: 6  
 Explanation: The subarray  $[4, -1, 2, 1]$  has the largest sum = 6.

### **Solution:**

```
function maxSubArray(nums) {
```

```

let maxSum = nums[0];
let currentSum = nums[0];

for (let i = 1; i < nums.length; i++) {
    currentSum = Math.max(nums[i], currentSum + nums[i]); // Add current element or start new subarray
    maxSum = Math.max(maxSum, currentSum); // Update maximum sum if needed
}

return maxSum;
}

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

```

### **Explanation:**

1. **Objective:** Find the subarray with the largest sum.
2. **Approach:** Use Kadane's algorithm:
  - o currentSum: Tracks the maximum sum of the subarray ending at the current index.
  - o maxSum: Stores the maximum sum found so far.
3. **Steps:**
  - o Initialize currentSum and maxSum with the first element.
  - o Iterate through the array starting from the second element:
    - Update currentSum to either the current element (nums[i]) or the sum of the current element with the previous subarray (currentSum + nums[i]), whichever is larger.
    - Update maxSum with the maximum of maxSum and currentSum.
4. **Why it Works:** Kadane's algorithm dynamically decides whether to include the current element in the existing subarray or start a new one based on the maximum sum.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Uses only two variables.

### **Maximum Subarray (Return Subarray Indices)**

**32. Problem: Find the contiguous subarray with the largest sum and return its starting and ending indices along with the sum.**

#### **Example:**

Input: nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]  
 Output: { start: 3, end: 6, maxSum: 6 }  
 Explanation: The subarray [4, -1, 2, 1] has the largest sum = 6, starting at index 3 and ending at index 6.

### Solution:

```
function maxSubArrayWithIndices(nums) {  
    let maxSum = nums[0];  
    let currentSum = nums[0];  
    let start = 0,  
        end = 0,  
        tempStart = 0;  
  
    for (let i = 1; i < nums.length; i++) {  
        if (nums[i] > currentSum + nums[i]) {  
            currentSum = nums[i];  
            tempStart = i; // Start a new subarray  
        } else {  
            currentSum += nums[i];  
        }  
  
        if (currentSum > maxSum) {  
            maxSum = currentSum;  
            start = tempStart;  
            end = i;  
        }  
    }  
  
    return { start, end, maxSum };  
}  
  
// Test  
console.log(maxSubArrayWithIndices([-2, 1, -3, 4, -1, 2, 1, -5, 4])); // Output: { start: 3, end: 6, maxSum: 6 }
```

### Explanation:

1. **Objective:** Identify the maximum subarray and its indices.
2. **Approach:** Modify Kadane's algorithm to track the starting and ending indices:
  - o Use tempStart to track the potential start of a new subarray.
  - o Update start and end when maxSum changes.
3. **Steps:**
  - o Initialize variables for the maximum sum, current sum, and indices.
  - o Iterate through the array:
    - If the current element is larger than the current sum with the element, start a new subarray.
    - Update the start and end indices when a new maximum sum is found.
4. **Why it Works:** Tracks the subarray bounds dynamically as the maximum sum evolves.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Uses constant additional space.

## Maximum Subarray (Divide and Conquer)

**33. Problem:** Use the divide-and-conquer strategy to find the maximum subarray sum.

**Example:**

```
Input: nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
Output: 6
Explanation: The subarray [4, -1, 2, 1] has the largest sum = 6.
```

**Solution:**

```
function maxSubArrayDivideAndConquer(nums) {
    function helper(left, right) {
        if (left === right) return nums[left]; // Base case: single element

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

        const leftMax = helper(left, mid); // Max in left half
        const rightMax = helper(mid + 1, right); // Max in right half

        let crossMax = nums[mid],
            tempSum = 0;
        for (let i = mid; i >= left; i--) {
            // Max sum crossing to the left
            tempSum += nums[i];
            crossMax = Math.max(crossMax, tempSum);
        }

        tempSum = crossMax;
        for (let i = mid + 1; i <= right; i++) {
            // Max sum crossing to the right
            tempSum += nums[i];
            crossMax = Math.max(crossMax, tempSum);
        }

        return Math.max(leftMax, rightMax, crossMax); // Max of left, right, and crossing
    }

    return helper(0, nums.length - 1);
}

// Test
console.log(maxSubArrayDivideAndConquer([-2, 1, -3, 4, -1, 2, 1, -5, 4])); // Output: 6
```

**Explanation:**

1. **Objective:** Divide the array into halves and solve for each half recursively.

2. **Approach:** For any given segment:
  - o Find the maximum subarray in the left half.
  - o Find the maximum subarray in the right half.
  - o Find the maximum subarray crossing the midpoint.
3. **Steps:**
  - o Recursively divide the array into smaller halves.
  - o Calculate the maximum sum of subarrays crossing the midpoint.
  - o Return the maximum of the three (left, right, crossing).
4. **Why it Works:** Divide-and-conquer ensures that all possible subarrays are considered in an efficient manner.
5. **Time Complexity:**  $O(n \log n)$  — Each division takes  $O(\log n)$ , and merging takes  $O(n)$ .
6. **Space Complexity:**  $O(\log n)$  — Recursive stack depth.

### Maximum Subarray (Dynamic Programming with Subarray)

**34. Problem:** **Find the maximum subarray sum and return the subarray itself, not just the sum.**

**Example:**

Input: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`  
 Output: { maxSum: 6, subarray: [4, -1, 2, 1] }  
 Explanation: The subarray [4, -1, 2, 1] has the largest sum = 6.

**Solution:**

```
function maxSubArrayWithSubarray(nums) {
  let maxSum = nums[0];
  let currentSum = nums[0];
  let start = 0,
    end = 0,
    tempStart = 0;

  for (let i = 1; i < nums.length; i++) {
    if (nums[i] > currentSum + nums[i]) {
      currentSum = nums[i];
      tempStart = i; // Start a new subarray
    } else {
      currentSum += nums[i];
    }

    if (currentSum > maxSum) {
      maxSum = currentSum;
      start = tempStart;
      end = i;
    }
  }
}
```

```

    return { maxSum, subarray: nums.slice(start, end + 1) };
}

// Test
console.log(maxSubArrayWithSubarray([-2, 1, -3, 4, -1, 2, 1, -5, 4])); // Output: { maxSum:
6, subarray: [4, -1, 2, 1] }

```

### **Explanation:**

1. **Objective:** Find both the maximum subarray sum and the actual subarray.
2. **Approach:** Modify Kadane's algorithm to track the start and end indices of the subarray.
3. **Steps:**
  - o Track the start of a potential new subarray with tempStart.
  - o Update the start and end indices when maxSum changes.
  - o Return both the maxSum and the subarray using nums.slice(start, end + 1).
4. **Why it Works:** By maintaining indices alongside the sum calculations, we can extract the subarray that contributes to the maximum sum.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Uses constant extra space.

### **Maximum Subarray (Prefix Sum)**

#### **35. Problem: Use the prefix sum technique to find the maximum subarray sum.**

##### **Example:**

Input: nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]  
Output: 6  
Explanation: The subarray [4, -1, 2, 1] has the largest sum = 6.

##### **Solution:**

```

function maxSubArrayPrefixSum(nums) {
  let prefixSum = 0;
  let minPrefixSum = 0;
  let maxSum = -Infinity;

  for (let num of nums) {
    prefixSum += num; // Accumulate the sum
    maxSum = Math.max(maxSum, prefixSum - minPrefixSum); // Calculate max subarray
    sum
    minPrefixSum = Math.min(minPrefixSum, prefixSum); // Update the minimum prefix sum
  }

  return maxSum;
}

```

```
// Test  
console.log(maxSubArrayPrefixSum([-2, 1, -3, 4, -1, 2, 1, -5, 4])); // Output: 6
```

### Explanation:

1. **Objective:** Use the prefix sum concept to calculate the maximum subarray sum efficiently.
2. **Approach:** Maintain two variables:
  - o prefixSum: The cumulative sum up to the current index.
  - o minPrefixSum: The smallest prefix sum encountered so far.
3. **Steps:**
  - o For each element, calculate the current prefix sum.
  - o Update maxSum as the difference between prefixSum and minPrefixSum (this represents the largest subarray sum ending at the current index).
  - o Update minPrefixSum if the current prefixSum is smaller.
4. **Why it Works:** The difference between the current prefix sum and the smallest prefix sum gives the maximum subarray sum ending at the current index.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Uses only three variables.

### Maximum Subarray (Circular Array)

**36. Problem:** Given a circular array, find the maximum subarray sum. A circular array means that the last element wraps around to the first element.

#### Example:

```
Input: nums = [5, -3, 5]  
Output: 10  
Explanation: The subarray [5, 5] (wrapping around) has the largest sum = 10.
```

#### Solution:

```
function maxSubArrayCircular(nums) {  
    let maxSum = nums[0],  
        currentMax = nums[0];  
    let minSum = nums[0],  
        currentMin = nums[0];  
    let totalSum = nums[0];  
  
    for (let i = 1; i < nums.length; i++) {  
        currentMax = Math.max(nums[i], currentMax + nums[i]); // Max subarray sum  
        maxSum = Math.max(maxSum, currentMax);  
  
        currentMin = Math.min(nums[i], currentMin + nums[i]); // Min subarray sum  
        minSum = Math.min(minSum, currentMin);  
  
        totalSum += nums[i];  
    }  
    return maxSum - minSum;  
}
```

```

    }

    return maxSum > 0 ? Math.max(maxSum, totalSum - minSum) : maxSum;
}

// Test
console.log(maxSubArrayCircular([5, -3, 5])); // Output: 10

```

### Explanation:

1. **Objective:** Handle wrap-around subarrays in circular arrays.
2. **Approach:**
  - o Use Kadane's algorithm to calculate:
    - maxSum: The maximum subarray sum in the regular array.
    - minSum: The minimum subarray sum in the regular array.
    - totalSum: The total sum of the array.
  - o The circular maximum is given by totalSum - minSum (excluding the minimum subarray).
  - o If all elements are negative, return maxSum (to avoid a zero result from totalSum - minSum).
3. **Steps:**
  - o Traverse the array to calculate maxSum, minSum, and totalSum.
  - o Return the maximum of maxSum and totalSum - minSum.
4. **Why it Works:** The wrap-around sum is equivalent to excluding the minimum subarray from the total sum.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Uses constant extra space.

### Maximum Subarray (Using Sliding Window)

#### 37. Problem: Find the maximum subarray sum using a sliding window approach.

##### Example:

```

Input: nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
Output: 6
Explanation: The subarray [4, -1, 2, 1] has the largest sum = 6.

```

##### Solution:

```

function maxSubArraySlidingWindow(nums) {
  let maxSum = nums[0];
  let currentSum = nums[0];

  for (let i = 1; i < nums.length; i++) {
    if (currentSum < 0) {
      currentSum = nums[i]; // Reset the window
    }
    currentSum += nums[i];
    maxSum = Math.max(maxSum, currentSum);
  }
  return maxSum;
}

```

```

} else {
    currentSum += nums[i]; // Expand the window
}

maxSum = Math.max(maxSum, currentSum);
}

return maxSum;
}

// Test
console.log(maxSubArraySlidingWindow([-2, 1, -3, 4, -1, 2, 1, -5, 4])); // Output: 6

```

### **Explanation:**

1. **Objective:** Use a sliding window to find the maximum subarray sum.
2. **Approach:** Dynamically adjust the window size:
  - o If the current sum becomes negative, reset the window.
  - o Otherwise, expand the window to include the next element.
3. **Steps:**
  - o Initialize currentSum and maxSum with the first element.
  - o Traverse the array:
    - Reset currentSum when it drops below zero.
    - Expand the window by adding the current element.
    - Update maxSum with the maximum of itself and currentSum.
4. **Why it Works:** By dynamically resetting the window, this approach ensures only positive contributions are considered for the maximum sum.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Uses only two variables.

### **Maximum Subarray (Product-Based Variations)**

**38. Problem: Find the contiguous subarray within an array (containing at least one number) which has the largest product.**

#### **Example:**

```

Input: nums = [2, 3, -2, 4]
Output: 6
Explanation: The subarray [2, 3] has the largest product = 6.

```

#### **Solution:**

```

function maxProductSubarray(nums) {
  let maxProduct = nums[0];
  let minProduct = nums[0];
  let result = nums[0];

```

```

for (let i = 1; i < nums.length; i++) {
    if (nums[i] < 0) {
        [maxProduct, minProduct] = [minProduct, maxProduct]; // Swap max and min for
negative numbers
    }

    maxProduct = Math.max(nums[i], maxProduct * nums[i]); // Maximum product including
nums[i]
    minProduct = Math.min(nums[i], minProduct * nums[i]); // Minimum product including
nums[i]

    result = Math.max(result, maxProduct);
}

return result;
}

// Test
console.log(maxProductSubarray([2, 3, -2, 4])); // Output: 6

```

### **Explanation:**

1. **Objective:** Find the contiguous subarray with the largest product.
2. **Approach:**
  - o Maintain maxProduct and minProduct for each iteration:
    - maxProduct: Maximum product including the current element.
    - minProduct: Minimum product including the current element (useful for handling negative numbers).
  - o Swap maxProduct and minProduct when encountering a negative number because the sign flips.
3. **Steps:**
  - o Initialize maxProduct, minProduct, and result with the first element.
  - o Traverse the array:
    - Swap maxProduct and minProduct if the current number is negative.
    - Update maxProduct and minProduct based on the current element.
    - Update result to track the global maximum product.
4. **Why it Works:** Tracking both maximum and minimum products ensures that negative numbers are handled correctly.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Uses only a few variables.

### **Maximum Subarray (Non-Contiguous)**

#### **39. Problem: Find the maximum sum of any subset (not necessarily contiguous) of the array.**

##### **Example:**

Input: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`

Output: 12

Explanation: The subset [4, 2, 1, 4] gives the largest sum = 12.

### Solution:

```
function maxNonContiguousSubarray(nums) {
    let maxSum = 0;
    let hasPositive = false;

    let maxNegative = -Infinity;

    for (let num of nums) {
        if (num > 0) {
            maxSum += num;
            hasPositive = true;
        } else {
            maxNegative = Math.max(maxNegative, num);
        }
    }

    return hasPositive ? maxSum : maxNegative; // If no positive numbers, return the largest negative number
}

// Test
console.log(maxNonContiguousSubarray([-2, 1, -3, 4, -1, 2, 1, -5, 4])); // Output: 12
console.log(maxNonContiguousSubarray([-2, -3, -1, -5])); // Output: -1
```

### Explanation:

1. **Objective:** Maximize the sum of a non-contiguous subset of the array.
2. **Approach:**
  - o Sum up all positive numbers since a non-contiguous subset allows skipping negative values.
  - o If no positive numbers exist, return the largest negative number.
3. **Steps:**
  - o Traverse the array:
    - Add positive numbers to the maxSum.
    - Track the largest negative number in case there are no positives.
  - o Return the sum of positive numbers or the largest negative number.
4. **Why it Works:** Non-contiguous subsets allow skipping negative values, so this greedy approach suffices.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Uses only a few variables.

### Maximum Subarray (Strictly Increasing Elements)

## 40. Problem: Find the maximum sum of a contiguous subarray where the elements are strictly increasing.

### Example:

Input: `nums = [1, 3, 5, 4, 7]`

Output: 9

Explanation: The subarray [1, 3, 5] has the largest sum = 9.

### Solution:

```
function maxIncreasingSubarray(nums) {  
    let maxSum = nums[0];  
    let currentSum = nums[0];  
  
    for (let i = 1; i < nums.length; i++) {  
        if (nums[i] > nums[i - 1]) {  
            currentSum += nums[i]; // Add to current sum if increasing  
        } else {  
            currentSum = nums[i]; // Reset current sum if not increasing  
        }  
  
        maxSum = Math.max(maxSum, currentSum); // Update maximum sum  
    }  
  
    return maxSum;  
}  
  
// Test  
console.log(maxIncreasingSubarray([1, 3, 5, 4, 7])); // Output: 9  
console.log(maxIncreasingSubarray([2, 2, 2, 2, 2])); // Output: 2
```

### Explanation:

1. **Objective:** Find the maximum sum of a strictly increasing subarray.
2. **Approach:** Use a dynamic sliding window:
  - o Accumulate the sum while the sequence is increasing.
  - o Reset the sum when the sequence stops increasing.
3. **Steps:**
  - o Initialize maxSum and currentSum with the first element.
  - o Traverse the array:
    - Add to currentSum if the current element is greater than the previous.
    - Otherwise, reset currentSum to the current element.
    - Update maxSum after each iteration.
4. **Why it Works:** By tracking increasing segments and resetting when necessary, this approach finds the desired subarray efficiently.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.

6. **Space Complexity:** O(1) — Uses only two variables.

## Contains Duplicate

### Contains Duplicate (Basic Check)

**41. Problem:** Given an integer array `nums`, return true if any value appears at least twice in the array, and false if every element is distinct.

**Example:**

```
Input: nums = [1, 2, 3, 1]
Output: true
Explanation: The number 1 appears twice.
```

```
Input: nums = [1, 2, 3, 4]
Output: false
Explanation: All elements are distinct.
```

**Solution:**

```
function containsDuplicate(nums) {
  const seen = new Set();

  for (let num of nums) {
    if (seen.has(num)) {
      return true; // Duplicate found
    }
    seen.add(num);
  }

  return false; // No duplicates found
}

// Test
console.log(containsDuplicate([1, 2, 3, 1])); // Output: true
console.log(containsDuplicate([1, 2, 3, 4])); // Output: false
```

**Explanation:**

1. **Objective:** Determine if any number appears more than once in the array.
2. **Approach:** Use a Set to track unique elements.
3. **Steps:**
  - o Iterate through the array.
  - o For each number:

- If it already exists in the Set, return true immediately.
  - Otherwise, add it to the Set.
- If the loop completes without finding duplicates, return false.
- Why it Works:** The Set data structure ensures constant-time lookups and insertions.
  - Time Complexity:**  $O(n)$  — Single traversal of the array.
  - Space Complexity:**  $O(n)$  — Space for the Set.

### Contains Duplicate II (Indices Difference Constraint)

**42. Problem: Return true if there are two distinct indices i and j in the array such that  $\text{nums}[i] == \text{nums}[j]$  and the absolute difference between i and j is at most k.**

**Example:**

Input:  $\text{nums} = [1, 2, 3, 1]$ ,  $k = 3$   
 Output: **true**  
 Explanation:  $\text{nums}[0] = \text{nums}[3] = 1$  and  $|0 - 3| \leq 3$ .

Input:  $\text{nums} = [1, 0, 1, 1]$ ,  $k = 1$   
 Output: **true**  
 Explanation:  $\text{nums}[2] = \text{nums}[3] = 1$  and  $|2 - 3| \leq 1$ .

**Solution:**

```
function containsNearbyDuplicate(nums, k) {
  const map = new Map();

  for (let i = 0; i < nums.length; i++) {
    if (map.has(nums[i]) && i - map.get(nums[i]) <= k) {
      return true; // Duplicate within range found
    }
    map.set(nums[i], i); // Update the index of the current number
  }

  return false; // No duplicates within range
}

// Test
console.log(containsNearbyDuplicate([1, 2, 3, 1], 3)); // Output: true
console.log(containsNearbyDuplicate([1, 0, 1, 1], 1)); // Output: true
console.log(containsNearbyDuplicate([1, 2, 3, 1, 2, 3], 2)); // Output: false
```

**Explanation:**

- Objective:** Check if a duplicate exists with an index difference of at most k.
- Approach:** Use a hash map to store the index of each element.
- Steps:**
  - Iterate through the array.

- For each number:
    - If it exists in the map and the index difference is less than or equal to k, return true.
    - Otherwise, update the map with the current index.
  - If no such duplicates are found, return false.
4. **Why it Works:** The map efficiently tracks the latest occurrence of each number, enabling constant-time index comparisons.
  5. **Time Complexity:** O(n) — Single traversal of the array.
  6. **Space Complexity:** O(n) — Space for the hash map.

### Contains Duplicate III (Value and Index Constraints)

**43. Problem:** Return true if there are two distinct indices i and j such that:

1.  $|nums[i] - nums[j]| \leq t$ , and
2.  $|i - j| \leq k$ .

**Example:**

Input: `nums = [1, 5, 9, 1]`,  $k = 2$ ,  $t = 3$

Output: `false`

Explanation: No two indices satisfy both conditions.

Input: `nums = [1, 2, 3, 1]`,  $k = 3$ ,  $t = 0$

Output: `true`

Explanation: `nums[0] = nums[3] = 1` and  $|0 - 3| \leq 3$ .

**Solution:**

```
function containsNearbyAlmostDuplicate(nums, k, t) {
  const bucket = new Map();
  const getBucketKey = (num, size) => Math.floor(num / size);

  if (t < 0) return false;

  for (let i = 0; i < nums.length; i++) {
    const bucketKey = getBucketKey(nums[i], t + 1);

    if (bucket.has(bucketKey)) return true;
    if (
      bucket.has(bucketKey - 1) &&
      Math.abs(nums[i] - bucket.get(bucketKey - 1)) <= t
    )
      return true;
    if (
      bucket.has(bucketKey + 1) &&
      Math.abs(nums[i] - bucket.get(bucketKey + 1)) <= t
    )
      bucket.set(bucketKey, nums[i]);
  }
}
```

```

        )
    return true;

    bucket.set(bucketKey, nums[i]);

    if (i >= k) {
        bucket.delete(getBucketKey(nums[i - k], t + 1));
    }
}

return false;
}

// Test
console.log(containsNearbyAlmostDuplicate([1, 5, 9, 1], 2, 3)); // Output: false
console.log(containsNearbyAlmostDuplicate([1, 2, 3, 1], 3, 0)); // Output: true

```

### **Explanation:**

1. **Objective:** Check if any two elements satisfy both the value and index constraints.
2. **Approach:** Use a bucket-based approach:
  - o Group numbers into "buckets" of size  $t + 1$ .
  - o Numbers in the same bucket are at most  $t$  apart.
  - o Adjacent buckets are checked for potential matches.
3. **Steps:**
  - o Calculate the bucket key for each number using the formula  $\text{Math.floor}(\text{num} / \text{size})$ .
  - o Check for duplicates in the same bucket or adjacent buckets.
  - o Maintain a sliding window of size  $k$  by removing numbers that fall out of range.
4. **Why it Works:** The bucket-based approach ensures efficient comparisons for both value and index constraints.
5. **Time Complexity:**  $O(n)$  — Each number is processed once.
6. **Space Complexity:**  $O(k)$  — At most  $k$  buckets are stored at any time.

### **Contains Duplicate (Sort and Compare)**

**44. Problem: Check if any value appears at least twice in the array by sorting the array first.**

#### **Example:**

Input:  $\text{nums} = [1, 2, 3, 1]$ ;  
Output: **true**;

Input:  $\text{nums} = [1, 2, 3, 4]$ ;  
Output: **false**;

### Solution:

```
function containsDuplicateWithSort(nums) {  
    nums.sort((a, b) => a - b); // Sort the array in ascending order  
  
    for (let i = 1; i < nums.length; i++) {  
        if (nums[i] === nums[i - 1]) {  
            return true; // Duplicate found  
        }  
    }  
  
    return false; // No duplicates found  
}  
  
// Test  
console.log(containsDuplicateWithSort([1, 2, 3, 1])); // Output: true  
console.log(containsDuplicateWithSort([1, 2, 3, 4])); // Output: false
```

### Explanation:

1. **Objective:** Determine if there are duplicates by sorting the array and comparing adjacent elements.
2. **Approach:**
  - o Sort the array.
  - o Traverse the sorted array and check if any adjacent elements are equal.
3. **Steps:**
  - o Use sort to arrange the elements in ascending order.
  - o Compare `nums[i]` with `nums[i - 1]` for all elements from index 1 onward.
  - o Return true if a match is found, otherwise return false.
4. **Why it Works:** Sorting places duplicate elements next to each other, allowing for a simple comparison.
5. **Time Complexity:**  $O(n \log n)$  — Due to sorting.
6. **Space Complexity:**  $O(1)$  (ignoring sorting space) — No additional data structures are used.

### Contains Duplicate (Count Frequency with Hash Map)

**45. Problem:** **Return true if there is any duplicate in the array by counting the frequency of each number.**

#### Example:

Input: `nums = [1, 2, 3, 1];`  
Output: `true;`

Input: `nums = [1, 2, 3, 4];`  
Output: `false;`

### Solution:

```
function containsDuplicateWithFrequency(nums) {  
    const frequency = {};  
  
    for (let num of nums) {  
        if (frequency[num]) {  
            return true; // Duplicate found  
        }  
        frequency[num] = 1; // Mark as seen  
    }  
  
    return false; // No duplicates found  
}  
  
// Test  
console.log(containsDuplicateWithFrequency([1, 2, 3, 1])); // Output: true  
console.log(containsDuplicateWithFrequency([1, 2, 3, 4])); // Output: false
```

### Explanation:

1. **Objective:** Track the frequency of each number in the array and detect duplicates.
2. **Approach:** Use an object (frequency) as a hash map to store the occurrence of each number.
3. **Steps:**
  - o Traverse the array.
  - o For each number, check if it exists in the frequency map:
    - If yes, return true.
    - Otherwise, set its value to 1 in the map.
  - o If the loop completes, return false.
4. **Why it Works:** The hash map allows for constant-time lookups and insertion.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(n)$  — Space for the hash map.

### Contains Duplicate (All Duplicate Values)

#### 46. Problem: Return an array of all duplicate values in the array.

##### Example:

Input: `nums = [1, 2, 3, 1, 2, 4];`  
Output: `[1, 2];`

Input: `nums = [1, 2, 3, 4];`  
Output: `[];`

### Solution:

```

function findAllDuplicates(nums) {
  const seen = new Set();
  const duplicates = new Set();

  for (let num of nums) {
    if (seen.has(num)) {
      duplicates.add(num); // Add to duplicates if already seen
    } else {
      seen.add(num); // Add to seen if not seen before
    }
  }

  return Array.from(duplicates); // Convert set to array
}

// Test
console.log(findAllDuplicates([1, 2, 3, 1, 2, 4])); // Output: [1, 2]
console.log(findAllDuplicates([1, 2, 3, 4])); // Output: []

```

### **Explanation:**

1. **Objective:** Identify all numbers that appear more than once in the array.
2. **Approach:** Use two sets:
  - o **seen:** Tracks numbers that have been encountered.
  - o **duplicates:** Tracks numbers that appear more than once.
3. **Steps:**
  - o Traverse the array:
    - If the current number is already in seen, add it to duplicates.
    - Otherwise, add it to seen.
  - o Convert the duplicates set to an array and return it.
4. **Why it Works:** The two sets efficiently separate unique numbers from duplicates.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(n)$  — Space for the two sets.

### **Contains Duplicate (Remove Duplicates In-Place)**

**47. Problem: Modify the array in-place to remove duplicate values and return the length of the modified array.**

### **Example:**

```

Input: nums = [1, 2, 3, 1, 2, 4]
Output: 4
Modified Array: [1, 2, 3, 4]

```

### **Solution:**

```

function removeDuplicates(nums) {
    nums.sort((a, b) => a - b); // Sort the array in-place

    let index = 0; // Pointer for unique elements

    for (let i = 1; i < nums.length; i++) {
        if (nums[i] !== nums[index]) {
            index++;
            nums[index] = nums[i]; // Move unique element to the next position
        }
    }

    return index + 1; // Return the number of unique elements
}

// Test
const nums = [1, 2, 3, 1, 2, 4];
const length = removeDuplicates(nums);
console.log(length); // Output: 4
console.log(nums.slice(0, length)); // Output: [1, 2, 3, 4]

```

### **Explanation:**

1. **Objective:** Remove duplicates while modifying the array in-place.
2. **Approach:**
  - o Sort the array to group duplicates together.
  - o Use a pointer (index) to track the position for unique elements.
3. **Steps:**
  - o Sort the array in ascending order.
  - o Traverse the array:
    - If the current element is different from the last unique element, move it to the next position (index + 1).
  - o Return index + 1 as the number of unique elements.
4. **Why it Works:** Sorting groups duplicates, and the pointer ensures minimal modifications.
5. **Time Complexity:**  $O(n \log n)$  — Due to sorting.
6. **Space Complexity:**  $O(1)$  — In-place modification.

### **Contains Duplicate (Find First Duplicate)**

**48. Problem:** **Return the first duplicate value that appears in the array. If there are no duplicates, return -1.**

### **Example:**

**Input:** `nums = [2, 1, 3, 5, 3, 2]`

Output: 3  
Explanation: The number 3 is the first duplicate to appear.

Input: `nums = [1, 2, 3, 4]`  
Output: -1  
Explanation: There are no duplicates.

### Solution:

```
function findFirstDuplicate(nums) {  
    const seen = new Set();  
  
    for (let num of nums) {  
        if (seen.has(num)) {  
            return num; // Return the first duplicate  
        }  
        seen.add(num);  
    }  
  
    return -1; // No duplicates found  
}  
  
// Test  
console.log(findFirstDuplicate([2, 1, 3, 5, 3, 2])); // Output: 3  
console.log(findFirstDuplicate([1, 2, 3, 4])); // Output: -1
```

### Explanation:

1. **Objective:** Identify the first duplicate value in the array.
2. **Approach:** Use a Set to track elements as they are encountered.
3. **Steps:**
  - o Traverse the array.
  - o For each number:
    - If it exists in the Set, return it as the first duplicate.
    - Otherwise, add it to the Set.
  - o If the loop completes without finding duplicates, return -1.
4. **Why it Works:** The Set ensures efficient lookups to detect the first repeated value.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(n)$  — Space for the Set.

### Contains Duplicate (Count Total Duplicates)

#### 49. Problem: **Return the total number of duplicate values in the array.**

### Example:

Input: `nums = [1, 2, 3, 1, 2, 4, 4]`

Output: 3

Explanation: The duplicates are 1, 2, and 4 (each counted once).

### Solution:

```
function countDuplicates(nums) {  
    const seen = new Set();  
    const duplicates = new Set();  
  
    for (let num of nums) {  
        if (seen.has(num)) {  
            duplicates.add(num); // Add to duplicates if already seen  
        } else {  
            seen.add(num); // Add to seen if not seen before  
        }  
    }  
  
    return duplicates.size; // Return the count of duplicate values  
}  
  
// Test  
console.log(countDuplicates([1, 2, 3, 1, 2, 4, 4])); // Output: 3  
console.log(countDuplicates([1, 2, 3, 4])); // Output: 0
```

### Explanation:

1. **Objective:** Count the total number of distinct duplicate values in the array.
2. **Approach:** Use two sets:
  - o **seen:** Tracks numbers that have been encountered.
  - o **duplicates:** Tracks numbers that appear more than once.
3. **Steps:**
  - o Traverse the array.
  - o For each number:
    - If it exists in seen, add it to duplicates.
    - Otherwise, add it to seen.
  - o Return the size of the duplicates set.
4. **Why it Works:** The duplicates set avoids counting the same duplicate multiple times.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(n)$  — Space for the two sets.

### Contains Duplicate (Bucket-Sort Approach)

**50. Problem:** Check if the array contains duplicates using a bucket-based approach for positive integers.

### Example:

Input: `nums = [1, 2, 3, 1];`

Output: `true;`

Input: `nums = [1, 2, 3, 4];`

Output: `false;`

### Solution:

```
function containsDuplicateBucket(nums) {  
    const bucket = new Array(Math.max(...nums) + 1).fill(false);  
  
    for (let num of nums) {  
        if (bucket[num]) {  
            return true; // Duplicate found  
        }  
        bucket[num] = true; // Mark as seen  
    }  
  
    return false; // No duplicates found  
}  
  
// Test  
console.log(containsDuplicateBucket([1, 2, 3, 1])); // Output: true  
console.log(containsDuplicateBucket([1, 2, 3, 4])); // Output: false
```

### Explanation:

1. **Objective:** Use a bucket-based approach to track occurrences of each number.
2. **Approach:**
  - o Create an array (bucket) of size equal to the maximum value in `nums` + 1.
  - o Use each number as an index in the bucket array to mark its presence.
3. **Steps:**
  - o Initialize a bucket array with false.
  - o Traverse the array:
    - If the bucket for the current number is already true, return true (duplicate found).
    - Otherwise, set the bucket value to true.
  - o If the loop completes without finding duplicates, return false.
4. **Why it Works:** The bucket array provides O(1) lookups for presence checks.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(max(nums)) — Space for the bucket array.

## Maximum Product Subarray

### Maximum Product Subarray (Basic Implementation)

## 51. Problem: Find the contiguous subarray within an array (containing at least one number) that has the largest product.

### Example:

Input: nums = [2, 3, -2, 4]

Output: 6

Explanation: The subarray [2, 3] has the largest product = 6.

Input: nums = [-2, 0, -1]

Output: 0

Explanation: The result is 0 because the subarray [0] has the largest product.

### Solution:

```
function maxProduct(nums) {  
    let maxProduct = nums[0];  
    let currentMax = nums[0];  
    let currentMin = nums[0];  
  
    for (let i = 1; i < nums.length; i++) {  
        if (nums[i] < 0) {  
            [currentMax, currentMin] = [currentMin, currentMax]; // Swap max and min  
        }  
  
        currentMax = Math.max(nums[i], currentMax * nums[i]);  
        currentMin = Math.min(nums[i], currentMin * nums[i]);  
  
        maxProduct = Math.max(maxProduct, currentMax);  
    }  
  
    return maxProduct;  
}  
  
// Test  
console.log(maxProduct([2, 3, -2, 4])); // Output: 6  
console.log(maxProduct([-2, 0, -1])); // Output: 0
```

### Explanation:

1. **Objective:** Find the contiguous subarray with the maximum product.

2. **Approach:**

- o Track currentMax and currentMin for each element, as negative numbers can flip the sign of the product.
- o Swap currentMax and currentMin when a negative number is encountered.

3. **Steps:**

- o Initialize maxProduct, currentMax, and currentMin to the first element.

- Iterate through the array:
    - Swap currentMax and currentMin if the current number is negative.
    - Update currentMax and currentMin using the current number.
    - Update maxProduct with the maximum of itself and currentMax.
4. **Why it Works:** Tracking both the maximum and minimum products ensures correct handling of negative numbers and zeroes.
  5. **Time Complexity:** O(n) — Single traversal of the array.
  6. **Space Complexity:** O(1) — Uses only a few variables.

### **Maximum Product Subarray (Return Subarray)**

**52. Problem: Find the contiguous subarray with the largest product and return the subarray itself.**

**Example:**

Input: nums = [2, 3, -2, 4]  
 Output: { maxProduct: 6, subarray: [2, 3] }

Input: nums = [-2, 0, -1]  
 Output: { maxProduct: 0, subarray: [0] }

**Solution:**

```
function maxProductWithSubarray(nums) {
  let maxProduct = nums[0];
  let currentMax = nums[0];
  let currentMin = nums[0];
  let start = 0,
    end = 0,
    tempStart = 0;

  for (let i = 1; i < nums.length; i++) {
    if (nums[i] < 0) {
      [currentMax, currentMin] = [currentMin, currentMax]; // Swap max and min
    }

    if (nums[i] > currentMax * nums[i]) {
      currentMax = nums[i];
      tempStart = i; // Start new subarray
    } else {
      currentMax *= nums[i];
    }

    currentMin = Math.min(nums[i], currentMin * nums[i]);

    if (currentMax > maxProduct) {
      maxProduct = currentMax;
    }
  }
}
```

```

        start = tempStart;
        end = i;
    }

    return { maxProduct, subarray: nums.slice(start, end + 1) };
}

// Test
console.log(maxProductWithSubarray([2, 3, -2, 4])); // Output: { maxProduct: 6, subarray: [2, 3] }
console.log(maxProductWithSubarray([-2, 0, -1])); // Output: { maxProduct: 0, subarray: [0] }

```

### Explanation:

1. **Objective:** Return both the maximum product and the subarray that produces it.
2. **Approach:**
  - o Use currentMax and currentMin as in the basic implementation.
  - o Track the starting and ending indices of the current subarray using tempStart, start, and end.
3. **Steps:**
  - o Swap currentMax and currentMin when encountering a negative number.
  - o Update currentMax and track the starting index of the subarray when a new subarray starts.
  - o Update maxProduct and the start and end indices when a new maximum is found.
4. **Why it Works:** Tracking indices alongside the product calculations ensures accurate identification of the subarray.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Uses constant additional space.

### Maximum Product Subarray (Handling Zeroes)

#### 53. Problem: Handle arrays that contain zeroes, which break the product continuity.

##### Example:

Input: nums = [0, 2, -3, 4, 0, -2, 1]  
Output: 4  
Explanation: The subarray [4] has the largest product.

Input: nums = [0, 0, -2, 0]  
Output: 0  
Explanation: The result is 0 as all other subarrays include zero.

##### Solution:

```

function maxProductWithZero(nums) {
    let maxProduct = nums[0];
    let currentMax = nums[0];
    let currentMin = nums[0];

    for (let i = 1; i < nums.length; i++) {
        if (nums[i] === 0) {
            currentMax = 0;
            currentMin = 0;
            maxProduct = Math.max(maxProduct, 0); // Handle zero
            continue;
        }

        if (nums[i] < 0) {
            [currentMax, currentMin] = [currentMin, currentMax]; // Swap max and min
        }

        currentMax = Math.max(nums[i], currentMax * nums[i]);
        currentMin = Math.min(nums[i], currentMin * nums[i]);

        maxProduct = Math.max(maxProduct, currentMax);
    }

    return maxProduct;
}

// Test
console.log(maxProductWithZero([0, 2, -3, 4, 0, -2, 1])); // Output: 4
console.log(maxProductWithZero([0, 0, -2, 0])); // Output: 0

```

### **Explanation:**

1. **Objective:** Handle subarrays that include zero, as zero resets the product.
2. **Approach:**
  - o Reset currentMax and currentMin to 0 whenever a 0 is encountered.
  - o Track the maximum product separately to include the effect of zeroes.
3. **Steps:**
  - o If the current element is 0, reset the products and update the maximum product.
  - o Otherwise, proceed as in the basic implementation.
4. **Why it Works:** Resetting ensures that zeroes do not disrupt calculations for subsequent subarrays.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Uses only a few variables.

### **Maximum Product Subarray (Divide and Conquer)**

#### 54. Problem: Use the divide-and-conquer strategy to find the maximum product subarray.

##### Example:

Input: nums = [2, 3, -2, 4]

Output: 6

Explanation: The subarray [2, 3] has the largest product.

Input: nums = [-2, 0, -1]

Output: 0

Explanation: The result is 0 because the subarray [0] has the largest product.

##### Solution:

```
function maxProductDivideAndConquer(nums, left = 0, right = nums.length - 1) {  
    if (left === right) return nums[left]; // Base case: single element  
  
    const mid = Math.floor((left + right) / 2);  
  
    // Max product in the left half  
    const leftMax = maxProductDivideAndConquer(nums, left, mid);  
  
    // Max product in the right half  
    const rightMax = maxProductDivideAndConquer(nums, mid + 1, right);  
  
    // Max product crossing the midpoint  
    let leftProduct = 1,  
        maxLeftProduct = -Infinity;  
    for (let i = mid; i >= left; i--) {  
        leftProduct *= nums[i];  
        maxLeftProduct = Math.max(maxLeftProduct, leftProduct);  
    }  
  
    let rightProduct = 1,  
        maxRightProduct = -Infinity;  
    for (let i = mid + 1; i <= right; i++) {  
        rightProduct *= nums[i];  
        maxRightProduct = Math.max(maxRightProduct, rightProduct);  
    }  
  
    const crossMax = maxLeftProduct * maxRightProduct;  
  
    return Math.max(leftMax, rightMax, crossMax); // Return the overall maximum  
}  
  
// Test  
console.log(maxProductDivideAndConquer([2, 3, -2, 4])); // Output: 6  
console.log(maxProductDivideAndConquer([-2, 0, -1])); // Output: 0
```

### **Explanation:**

1. **Objective:** Solve the problem recursively by dividing the array into smaller parts and finding the maximum product in each part.
2. **Approach:**
  - o Divide the array into two halves using the midpoint.
  - o Solve the problem for the left and right halves recursively.
  - o Find the maximum product that crosses the midpoint.
3. **Steps:**
  - o Calculate the product of subarrays crossing the midpoint by iterating outward from the middle.
  - o Return the maximum of the left half, right half, and the cross product.
4. **Why it Works:** Divide-and-conquer ensures all possible subarrays are considered by combining results from smaller subproblems.
5. **Time Complexity:**  $O(n \log n)$  — Recursive division and linear merging at each level.
6. **Space Complexity:**  $O(\log n)$  — Recursive stack depth.

### **Maximum Product Subarray (Dynamic Programming)**

**55. Problem: Find the maximum product subarray using a dynamic programming approach.**

#### **Example:**

Input: `nums = [2, 3, -2, 4]`

Output: 6

Explanation: The subarray [2, 3] has the largest product.

Input: `nums = [-2, 0, -1]`

Output: 0

Explanation: The result is 0 because the subarray [0] has the largest product.

#### **Solution:**

```
function maxProductDP(nums) {  
    const dpMax = Array(nums.length).fill(0);  
    const dpMin = Array(nums.length).fill(0);  
  
    dpMax[0] = nums[0];  
    dpMin[0] = nums[0];  
    let maxProduct = nums[0];  
  
    for (let i = 1; i < nums.length; i++) {  
        dpMax[i] = Math.max(  
            nums[i],  
            dpMax[i - 1] * nums[i],  
            dpMin[i - 1] * nums[i]  
        );  
        dpMin[i] = Math.min(  
            nums[i],  
            dpMax[i - 1] * nums[i],  
            dpMin[i - 1] * nums[i]  
        );  
        maxProduct = Math.max(maxProduct, dpMax[i]);  
    }  
    return maxProduct;  
}
```

```

dpMin[i] = Math.min(
    nums[i],
    dpMax[i - 1] * nums[i],
    dpMin[i - 1] * nums[i]
);
maxProduct = Math.max(maxProduct, dpMax[i]);
}

return maxProduct;
}

// Test
console.log(maxProductDP([2, 3, -2, 4])); // Output: 6
console.log(maxProductDP([-2, 0, -1])); // Output: 0

```

### **Explanation:**

1. **Objective:** Use dynamic programming to track the maximum and minimum products at each index.
2. **Approach:**
  - o Maintain two DP arrays:
    - $dpMax[i]$ : Maximum product ending at index  $i$ .
    - $dpMin[i]$ : Minimum product ending at index  $i$ .
  - o Update these arrays dynamically based on the current element.
3. **Steps:**
  - o Initialize the first element of both arrays as the first element of the input array.
  - o For each subsequent element, calculate the maximum and minimum products based on the previous values.
  - o Track the global maximum product during the iteration.
4. **Why it Works:** Tracking both maximum and minimum products ensures correct handling of negative numbers and zeroes.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(n)$  — Space for the two DP arrays.

### **Maximum Product Subarray (Single Pass Optimized)**

**56. Problem:** Optimize the solution to reduce space usage while maintaining linear time complexity.

### **Example:**

Input:  $nums = [2, 3, -2, 4]$ ;  
Output: 6;

Input:  $nums = [-2, 0, -1]$ ;  
Output: 0;

### **Solution:**

```

function maxProductOptimized(nums) {
    let currentMax = nums[0];
    let currentMin = nums[0];
    let maxProduct = nums[0];

    for (let i = 1; i < nums.length; i++) {
        if (nums[i] < 0) {
            [currentMax, currentMin] = [currentMin, currentMax]; // Swap max and min for negative numbers
        }

        currentMax = Math.max(nums[i], currentMax * nums[i]);
        currentMin = Math.min(nums[i], currentMin * nums[i]);

        maxProduct = Math.max(maxProduct, currentMax);
    }

    return maxProduct;
}

// Test
console.log(maxProductOptimized([2, 3, -2, 4])); // Output: 6
console.log(maxProductOptimized([-2, 0, -1])); // Output: 0

```

### **Explanation:**

1. **Objective:** Reduce space complexity by reusing variables instead of using additional arrays.
2. **Approach:** Use three variables:
  - o currentMax: Maximum product ending at the current index.
  - o currentMin: Minimum product ending at the current index.
  - o maxProduct: Global maximum product.
3. **Steps:**
  - o Initialize all variables to the first element of the array.
  - o Iterate through the array, updating currentMax, currentMin, and maxProduct dynamically.
  - o Swap currentMax and currentMin when encountering a negative number.
4. **Why it Works:** This approach eliminates the need for extra arrays, saving space while maintaining correctness.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Only a few variables are used.

### **Maximum Product Subarray (Handling All Negative Values)**

**57. Problem: Handle arrays where all numbers are negative, and return the maximum product subarray.**

### **Example:**

Input: `nums = [-1, -2, -3, -4]`

Output: 24

Explanation: The subarray `[-1, -2, -3, -4]` has the largest product = 24.

### Solution:

```
function maxProductAllNegatives(nums) {  
    let maxProduct = nums[0];  
    let currentMax = nums[0];  
    let currentMin = nums[0];  
  
    for (let i = 1; i < nums.length; i++) {  
        if (nums[i] < 0) {  
            [currentMax, currentMin] = [currentMin, currentMax]; // Swap max and min for negative  
            numbers  
        }  
  
        currentMax = Math.max(nums[i], currentMax * nums[i]);  
        currentMin = Math.min(nums[i], currentMin * nums[i]);  
  
        maxProduct = Math.max(maxProduct, currentMax);  
    }  
  
    return maxProduct;  
}  
  
// Test  
console.log(maxProductAllNegatives([-1, -2, -3, -4])); // Output: 24  
console.log(maxProductAllNegatives([-1, -2, -3, 0])); // Output: 6
```

### Explanation:

1. **Objective:** Handle cases where the array contains all negative numbers.
2. **Approach:**
  - o Use the same logic as the optimized solution for the general case.
  - o Maintain `currentMax` and `currentMin` to handle the product flipping when multiplying by negative numbers.
3. **Steps:**
  - o Swap `currentMax` and `currentMin` whenever a negative number is encountered.
  - o Update `maxProduct` with the maximum of itself and `currentMax`.
4. **Why it Works:** This approach dynamically adjusts for negative values, ensuring correct handling of flipping signs.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Uses only a few variables

### Maximum Product Subarray (With Zero as a Split Point)

## 58. Problem: Handle cases where zeros split the array into multiple subarrays, and find the maximum product.

### Example:

Input: `nums = [2, 3, 0, -2, 4, -1]`

Output: 8

Explanation: The subarray [-2, 4] has the largest product = 8.

Input: `nums = [0, -2, -3, 0, -4, -5]`

Output: 20

Explanation: The subarray [-4, -5] has the largest product = 20.

### Solution:

```
function maxProductWithZeroSplit(nums) {  
    let maxProduct = -Infinity;  
    let currentMax = 1;  
    let currentMin = 1;  
  
    for (let num of nums) {  
        if (num === 0) {  
            currentMax = 1; // Reset for next subarray  
            currentMin = 1;  
            maxProduct = Math.max(maxProduct, 0); // Include zero as a product  
            continue;  
        }  
  
        const temp = currentMax;  
        currentMax = Math.max(num, currentMax * num, currentMin * num);  
        currentMin = Math.min(num, temp * num, currentMin * num);  
  
        maxProduct = Math.max(maxProduct, currentMax);  
    }  
  
    return maxProduct;  
}  
  
// Test  
console.log(maxProductWithZeroSplit([2, 3, 0, -2, 4, -1])); // Output: 8  
console.log(maxProductWithZeroSplit([0, -2, -3, 0, -4, -5])); // Output: 20
```

### Explanation:

1. **Objective:** Handle cases where zeros break the array into smaller subarrays.
2. **Approach:**
  - o Treat zeros as split points by resetting `currentMax` and `currentMin` to 1.

- Keep track of the global maxProduct to include zero as a valid product.
3. **Steps:**
- Iterate through the array:
    - Reset products to 1 when encountering 0.
    - Update currentMax and currentMin for non-zero elements.
    - Update maxProduct after each iteration.
4. **Why it Works:** Resetting at zero ensures the calculation of independent subarrays, while maxProduct tracks the global maximum.
5. **Time Complexity:**  $O(n)$  — Single traversal of the array.
6. **Space Complexity:**  $O(1)$  — Uses only a few variables.

### **Maximum Product Subarray (Using Prefix and Suffix Products)**

#### **59. Problem: Use prefix and suffix product arrays to calculate the maximum product.**

##### **Example:**

Input: `nums = [2, 3, -2, 4]`

Output: 6

Explanation: The subarray [2, 3] has the largest product.

Input: `nums = [-2, 0, -1]`

Output: 0

Explanation: The subarray [0] has the largest product.

##### **Solution:**

```
function maxProductWithPrefixSuffix(nums) {
  let maxProduct = nums[0];
  let prefixProduct = 1;
  let suffixProduct = 1;

  for (let i = 0; i < nums.length; i++) {
    prefixProduct = (prefixProduct || 1) * nums[i];
    suffixProduct = (suffixProduct || 1) * nums[nums.length - 1 - i];
    maxProduct = Math.max(maxProduct, prefixProduct, suffixProduct);
  }

  return maxProduct;
}

// Test
console.log(maxProductWithPrefixSuffix([2, 3, -2, 4])); // Output: 6
console.log(maxProductWithPrefixSuffix([-2, 0, -1])); // Output: 0
```

##### **Explanation:**

1. **Objective:** Use prefix and suffix products to find the maximum product subarray.
2. **Approach:**
  - o Calculate the prefix product (left to right) and suffix product (right to left).
  - o Keep track of the maximum product encountered during both traversals.
3. **Steps:**
  - o Initialize prefixProduct and suffixProduct to 1.
  - o Iterate through the array:
    - Update prefixProduct using the current element.
    - Update suffixProduct using the reverse index.
    - Track the maximum of all products.
4. **Why it Works:** Prefix and suffix products ensure all possible subarrays are considered, including those split by zeros.
5. **Time Complexity:**  $O(n)$  — Two traversals of the array.
6. **Space Complexity:**  $O(1)$  — Uses only a few variables.

### **Maximum Product Subarray (Absolute Maximum Product)**

**60. Problem:** **Find the absolute maximum product subarray, including the largest negative product in case all numbers are negative.**

**Example:**

Input: `nums = [-1, -2, -3, -4]`

Output: 24

Explanation: The subarray `[-1, -2, -3, -4]` has the absolute maximum product = 24.

**Solution:**

```
function maxAbsoluteProduct(nums) {
  let currentMax = nums[0];
  let currentMin = nums[0];
  let maxProduct = nums[0];

  for (let i = 1; i < nums.length; i++) {
    if (nums[i] < 0) {
      [currentMax, currentMin] = [currentMin, currentMax];
    }

    currentMax = Math.max(nums[i], currentMax * nums[i]);
    currentMin = Math.min(nums[i], currentMin * nums[i]);

    maxProduct = Math.max(
      maxProduct,
      Math.abs(currentMax),
      Math.abs(currentMin)
    );
  }
}
```

```

    return maxProduct;
}

// Test
console.log(maxAbsoluteProduct([-1, -2, -3, -4])); // Output: 24
console.log(maxAbsoluteProduct([-2, -3, 4, -5])); // Output: 60

```

### **Explanation:**

1. **Objective:** Handle cases where the absolute maximum product may involve negative values.
2. **Approach:**
  - o Use Math.abs on currentMax and currentMin to track the absolute maximum product.
3. **Steps:**
  - o Swap currentMax and currentMin for negative numbers.
  - o Track the maximum absolute value of the product at each step.
4. **Why it Works:** Incorporating absolute values ensures that the magnitude of the product is maximized.
5. **Time Complexity:** O(n) — Single traversal of the array.
6. **Space Complexity:** O(1) — Uses only a few variables.

### **Search in Rotated Sorted Array:**

#### **Search in Rotated Sorted Array (No Duplicates)**

**61. Problem:** Given a sorted array that has been rotated at some unknown pivot and a target value, return the index of the target if it exists. Otherwise, return -1. Assume no duplicate values.

#### **Example:**

Input: nums = [4, 5, 6, 7, 0, 1, 2], target = 0

Output: 4

Explanation: The target 0 is at index 4.

Input: nums = [4, 5, 6, 7, 0, 1, 2], target = 3

Output: -1

Explanation: The target 3 does not exist in the array.

#### **Solution:**

```

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

  while (left <= right) {

```

```

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

if (nums[mid] === target) {
    return mid; // Target found
}

// 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
}

// Test
console.log(searchInRotatedArray([4, 5, 6, 7, 0, 1, 2], 0)); // Output: 4
console.log(searchInRotatedArray([4, 5, 6, 7, 0, 1, 2], 3)); // Output: -1

```

### Explanation:

1. **Objective:** Locate the target element in a rotated sorted array using binary search.
2. **Approach:**
  - o Use binary search to identify which half of the array is sorted.
  - o Narrow the search space based on the sorted half and the target value.
3. **Steps:**
  - o If the left half is sorted:
    - Check if the target is within the range of the left half and adjust pointers accordingly.
  - o Otherwise, the right half must be sorted:
    - Check if the target is within the range of the right half and adjust pointers.
4. **Why it Works:** Binary search leverages the sorted half to reduce the search space, even in a rotated array.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No additional space used.

## Search in Rotated Sorted Array (With Duplicates)

**62. Problem:** Handle cases where the array contains duplicate values and return the index of the target if it exists. Otherwise, return -1.

**Example:**

Input: (nums = [2, 5, 6, 0, 0, 1, 2]), (target = 0);  
Output: 3;

Input: (nums = [2, 5, 6, 0, 0, 1, 2]), (target = 3);  
Output: -1;

**Solution:**

```
function searchInRotatedArrayWithDuplicates(nums, target) {  
    let left = 0;  
    let right = nums.length - 1;  
  
    while (left <= right) {  
        const mid = Math.floor((left + right) / 2);  
  
        if (nums[mid] === target) {  
            return mid; // Target found  
        }  
  
        // Handle duplicates  
        if (nums[left] === nums[mid] && nums[mid] === nums[right]) {  
            left++;  
            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 -1; // Target not found
    }

// Test
console.log(searchInRotatedArrayWithDuplicates([2, 5, 6, 0, 0, 1, 2], 0)); // Output: 3
console.log(searchInRotatedArrayWithDuplicates([2, 5, 6, 0, 0, 1, 2], 3)); // Output: -1

```

### **Explanation:**

1. **Objective:** Handle duplicate values while searching for the target.
2. **Approach:**
  - o Add an extra step to handle duplicates by skipping over elements equal to `nums[mid]`.
  - o Continue with the logic for identifying the sorted half and narrowing the search space.
3. **Steps:**
  - o If duplicates are encountered (e.g., `nums[left] === nums[mid] === nums[right]`), increment left and decrement right to skip them.
  - o Proceed with binary search as in the no-duplicate case.
4. **Why it Works:** Removing duplicate elements reduces the search space while maintaining correctness.
5. **Time Complexity:**  $O(n)$  in the worst case (when all elements are duplicates), but  $O(\log n)$  on average.
6. **Space Complexity:**  $O(1)$  — No additional space used.

### **Search in Rotated Sorted Array (Find Pivot Index)**

#### **63. Problem: Find the pivot index (the index where the array is rotated) in a rotated sorted array.**

##### **Example:**

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`  
Output: 4  
Explanation: The pivot index is 4, where the smallest element (0) is located.

Input: `nums = [6, 7, 0, 1, 2, 3, 4]`  
Output: 2  
Explanation: The pivot index is 2, where the smallest element (0) is located.

### **Solution:**

```

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

    while (left < right) {

```

```

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

if (nums[mid] > nums[right]) {
    left = mid + 1; // Pivot is in the right half
} else {
    right = mid; // Pivot is in the left half or at mid
}
}

return left; // Pivot index
}

// Test
console.log(findPivotIndex([4, 5, 6, 7, 0, 1, 2])); // Output: 4
console.log(findPivotIndex([6, 7, 0, 1, 2, 3, 4])); // Output: 2

```

### Explanation:

1. **Objective:** Locate the index of the smallest element in the rotated sorted array, which is the pivot.
2. **Approach:**
  - o Use binary search to identify the pivot index:
    - Compare `nums[mid]` with `nums[right]` to decide which half of the array contains the pivot.
  - o Narrow the search space to the half containing the pivot.
3. **Steps:**
  - o If `nums[mid] > nums[right]`, the pivot is in the right half.
  - o Otherwise, the pivot is in the left half or at mid.
4. **Why it Works:** The pivot divides the rotated array into two sorted halves, and binary search can efficiently locate this division point.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No additional space used

### *Search in Rotated Sorted Array (Using Pivot Index)*

#### 64. Problem: Use the pivot index to search for a target value in a rotated sorted array.

##### Example:

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`, `target = 0`

Output: 4

Explanation: The target 0 is at index 4.

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`, `target = 3`

Output: -1

Explanation: The target 3 does not exist in the array.

##### Solution:

```

function searchUsingPivot(nums, target) {
    function findPivotIndex(nums) {
        let left = 0,
            right = nums.length - 1;
        while (left < right) {
            const mid = Math.floor((left + right) / 2);
            if (nums[mid] > nums[right]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    }

    const pivot = findPivotIndex(nums);
    let left = 0,
        right = nums.length - 1;

    // Determine the search bounds
    if (target >= nums[pivot] && target <= nums[right]) {
        left = pivot;
    } else {
        right = pivot - 1;
    }

    // Standard binary search
    while (left <= right) {
        const mid = Math.floor((left + right) / 2);
        if (nums[mid] === target) {
            return mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1; // Target not found
}

// Test
console.log(searchUsingPivot([4, 5, 6, 7, 0, 1, 2], 0)); // Output: 4
console.log(searchUsingPivot([4, 5, 6, 7, 0, 1, 2], 3)); // Output: -1

```

### Explanation:

1. **Objective:** Find the target value using the pivot index to divide the search space.

## 2. Approach:

- o Use the findPivotIndex function to locate the smallest element (pivot).
- o Based on the pivot, determine which half of the array to search.
- o Perform binary search within the determined bounds.

## 3. Steps:

- o Locate the pivot index using binary search.
- o If the target is within the range of elements after the pivot, search that range.
- o Otherwise, search the range before the pivot.

4. Why it Works: Dividing the array into two sorted halves ensures that binary search remains efficient.

5. Time Complexity:  $O(\log n)$  — Two binary searches: one to find the pivot and one to find the target.

6. Space Complexity:  $O(1)$  — No additional space used.

## Search in Rotated Sorted Array (Recursive)

**65. Problem: Implement a recursive solution to search for a target value in a rotated sorted array.**

### Example:

Input: (nums = [4, 5, 6, 7, 0, 1, 2]), (target = 1);  
Output: 5;

Input: (nums = [4, 5, 6, 7, 0, 1, 2]), (target = 3);  
Output: -1;

### Solution:

```
function searchRecursive(nums, target, left = 0, right = nums.length - 1) {  
    if (left > right) return -1;  
  
    const mid = Math.floor((left + right) / 2);  
  
    if (nums[mid] === target) {  
        return mid; // Target found  
    }  
  
    // Left half is sorted  
    if (nums[left] <= nums[mid]) {  
        if (nums[left] <= target && target < nums[mid]) {  
            return searchRecursive(nums, target, left, mid - 1);  
        } else {  
            return searchRecursive(nums, target, mid + 1, right);  
        }  
    }  
    // Right half is sorted  
    else {
```

```

if (nums[mid] < target && target <= nums[right]) {
    return searchRecursive(nums, target, mid + 1, right);
} else {
    return searchRecursive(nums, target, left, mid - 1);
}

// Test
console.log(searchRecursive([4, 5, 6, 7, 0, 1, 2], 1)); // Output: 5
console.log(searchRecursive([4, 5, 6, 7, 0, 1, 2], 3)); // Output: -1

```

### Explanation:

1. **Objective:** Implement a recursive version of binary search for a rotated sorted array.
2. **Approach:**
  - o Base Case: If left > right, return -1 (target not found).
  - o Check if the left or right half is sorted and recurse accordingly.
3. **Steps:**
  - o Compare nums[mid] with the target.
  - o If the left half is sorted, check if the target lies within its range:
    - If yes, recurse on the left half.
    - Otherwise, recurse on the right half.
  - o Repeat the logic for the right half.
4. **Why it Works:** Recursively dividing the search space ensures efficient exploration of the rotated array.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(\log n)$  — Recursive stack depth.

### Search in Rotated Sorted Array (Find Element in Circular Array)

#### 66. Problem: Find the target value in a circularly sorted array (similar to a rotated sorted array).

##### Example:

Input: nums = [10, 15, 20, 1, 3, 5, 8], target = 5

Output: 5

Explanation: The target 5 is at index 5.

Input: nums = [10, 15, 20, 1, 3, 5, 8], target = 25

Output: -1

Explanation: The target 25 does not exist in the array.

##### Solution:

```
function searchInCircularArray(nums, target) {
```

```

let left = 0;
let right = nums.length - 1;

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

    if (nums[mid] === target) {
        return mid; // Target found
    }

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

return -1; // Target not found
}

// Test
console.log(searchInCircularArray([10, 15, 20, 1, 3, 5, 8], 5)); // Output: 5
console.log(searchInCircularArray([10, 15, 20, 1, 3, 5, 8], 25)); // Output: -1

```

### Explanation:

1. **Objective:** Locate the target in a circularly sorted array using binary search.
2. **Approach:**
  - o Identify the sorted half of the array and narrow the search space accordingly.
3. **Steps:**
  - o Compare the middle element with the target.
  - o If the left half is sorted, check if the target lies within its range:
    - If yes, adjust right.
    - Otherwise, adjust left.
  - o Repeat the process for the right half if it is sorted.
4. **Why it Works:** Circular arrays can still be divided into two sorted halves, allowing binary search to function efficiently.

5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No additional space used.

### Search in Rotated Sorted Array (Return Closest Element)

**67. Problem: Find the closest element to a target value in a rotated sorted array.**

**Example:**

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`, `target = 3`

Output: 4

Explanation: The closest element to 3 is 4.

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`, `target = 8`

Output: 7

Explanation: The closest element to 8 is 7.

**Solution:**

```
function searchClosestInRotatedArray(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  let closest = nums[0];

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

    // Update the closest element if needed
    if (Math.abs(nums[mid] - target) < Math.abs(closest - target)) {
      closest = nums[mid];
    }

    if (nums[mid] === target) {
      return nums[mid]; // Target found
    }

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

```

        left = mid + 1; // Search in the right half
    } else {
        right = mid - 1; // Search in the left half
    }
}

return closest; // Return the closest element
}

// Test
console.log(searchClosestInRotatedArray([4, 5, 6, 7, 0, 1, 2], 3)); // Output: 4
console.log(searchClosestInRotatedArray([4, 5, 6, 7, 0, 1, 2], 8)); // Output: 7

```

### **Explanation:**

1. **Objective:** Find the element closest to the target if the target is not found.
2. **Approach:**
  - o Use binary search to traverse the rotated array.
  - o Update the closest element whenever a closer value is found.
3. **Steps:**
  - o Compare the target with the middle element.
  - o If the target is not found, update the closest value based on absolute difference.
  - o Adjust the search range based on the sorted half, as in standard binary search for rotated arrays.
4. **Why it Works:** Keeping track of the closest element during the binary search ensures we find the best possible match.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No additional space used.

### **Search in Rotated Sorted Array (Count Target Occurrences)**

#### **68. Problem: Count how many times a target value appears in a rotated sorted array.**

##### **Example:**

Input: (nums = [4, 5, 6, 7, 0, 1, 2, 4]), (target = 4);  
Output: 2;

Input: (nums = [4, 5, 6, 7, 0, 1, 2]), (target = 3);  
Output: 0;

##### **Solution:**

```
function countTargetOccurrences(nums, target) {
    let count = 0;
```

```

let left = 0;
let right = nums.length - 1;

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

    if (nums[mid] === target) {
        count++;
        // Count duplicates on the left and right
        let l = mid - 1;
        let r = mid + 1;
        while (l >= left && nums[l] === target) {
            count++;
            l--;
        }
        while (r <= right && nums[r] === target) {
            count++;
            r++;
        }
        break;
    }

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

return count;
}

// Test
console.log(countTargetOccurrences([4, 5, 6, 7, 0, 1, 2, 4], 4)); // Output: 2
console.log(countTargetOccurrences([4, 5, 6, 7, 0, 1, 2], 3)); // Output: 0

```

### **Explanation:**

1. **Objective:** Count all occurrences of the target in a rotated sorted array.
2. **Approach:**
  - o Use binary search to locate one occurrence of the target.
  - o Expand outward to count all duplicates around the found index.
3. **Steps:**
  - o Use binary search to find one occurrence of the target.
  - o Use two pointers to count duplicates to the left and right of the found index.
4. **Why it Works:** Binary search narrows the search range, and expanding outward ensures all duplicates are counted.
5. **Time Complexity:**  $O(\log n + k)$  — Binary search plus scanning for duplicates, where  $k$  is the number of duplicates.
6. **Space Complexity:**  $O(1)$  — No additional space used.

### **Search in Rotated Sorted Array (Find Minimum Value)**

**69. Problem:** **Find the minimum value in a rotated sorted array.**

#### **Example:**

```
Input: nums = [4, 5, 6, 7, 0, 1, 2];
Output: 0;
```

```
Input: nums = [6, 7, 8, 1, 2, 3, 4];
Output: 1;
```

#### **Solution:**

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

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

        if (nums[mid] > nums[right]) {
            left = mid + 1; // Minimum is in the right half
        } else {
            right = mid; // Minimum is in the left half or at mid
        }
    }

    return nums[left]; // The minimum value
}

// Test
console.log(findMinInRotatedArray([4, 5, 6, 7, 0, 1, 2])); // Output: 0
```

```
console.log(findMinInRotatedArray([6, 7, 8, 1, 2, 3, 4])); // Output: 1
```

### Explanation:

1. **Objective:** Find the smallest element in a rotated sorted array.
2. **Approach:** Use binary search to locate the pivot point, which corresponds to the minimum value.
3. **Steps:**
  - o Compare `nums[mid]` with `nums[right]`:
    - If `nums[mid] > nums[right]`, the minimum lies in the right half.
    - Otherwise, the minimum lies in the left half or at mid.
  - o Narrow the search space until left and right converge.
4. **Why it Works:** The pivot divides the rotated array into two sorted halves, and binary search efficiently narrows the range.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No additional space used.

### Search in Rotated Sorted Array (Find Maximum Value)

#### 70. Problem: **Find the maximum value in a rotated sorted array.**

##### Example:

```
Input: nums = [4, 5, 6, 7, 0, 1, 2];  
Output: 7;
```

```
Input: nums = [6, 7, 8, 1, 2, 3, 4];  
Output: 8;
```

##### Solution:

```
function findMaxInRotatedArray(nums) {  
    let left = 0;  
    let right = nums.length - 1;  
  
    while (left < right) {  
        const mid = Math.floor((left + right + 1) / 2); // Bias toward the right  
  
        if (nums[mid] > nums[left]) {  
            left = mid; // Maximum is in the right half or at mid  
        } else {  
            right = mid - 1; // Maximum is in the left half  
        }  
    }  
}
```

```

        return nums[left]; // The maximum value
    }

// Test
console.log(findMaxInRotatedArray([4, 5, 6, 7, 0, 1, 2])); // Output: 7
console.log(findMaxInRotatedArray([6, 7, 8, 1, 2, 3, 4])); // Output: 8

```

### **Explanation:**

1. **Objective:** Find the largest element in a rotated sorted array.
2. **Approach:** Use binary search with a slight modification to bias the search toward the right.
3. **Steps:**
  - o Compare  $\text{nums}[\text{mid}]$  with  $\text{nums}[\text{left}]$ :
    - If  $\text{nums}[\text{mid}] > \text{nums}[\text{left}]$ , the maximum lies in the right half or at  $\text{mid}$ .
    - Otherwise, the maximum lies in the left half.
  - o Narrow the search space until left and right converge.
4. **Why it Works:** Binary search efficiently identifies the largest element by leveraging the sorted nature of the array.
5. **Time Complexity:**  $O(\log n)$  — Binary search.
6. **Space Complexity:**  $O(1)$  — No additional space used.

## **Multi-Sum Combinations in Arrays:**

### **Multi-Sum Combinations in Arrays (Unique Triplets)**

**71. Problem: Given an integer array  $\text{nums}$ , return all unique triplets  $[\text{nums}[i], \text{nums}[j], \text{nums}[k]]$  such that:**

1.  $i \neq j, i \neq k$ , and  $j \neq k$
2.  $\text{nums}[i] + \text{nums}[j] + \text{nums}[k] == 0$

### **Example:**

Input:  $\text{nums} = [-1, 0, 1, 2, -1, -4]$ ;

Output: [  
 $[-1, -1, 2]$ ,  
 $[-1, 0, 1]$ ,  
 $]$ ;

Input:  $\text{nums} = []$ ;

Output: [];

Input:  $\text{nums} = [0]$ ;

Output: [];

### Solution:

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

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

        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]]);

                // Move pointers to skip duplicates
                while (left < right && nums[left] === nums[left + 1]) left++;
                while (left < right && nums[right] === nums[right - 1]) right--;

                left++;
                right--;
            } else if (sum < 0) {
                left++; // Increase sum
            } else {
                right--; // Decrease sum
            }
        }
    }

    return result;
}

// Test
console.log(threeSum([-1, 0, 1, 2, -1, -4])); // Output: [[-1, -1, 2], [-1, 0, 1]]
console.log(threeSum([])); // Output: []
console.log(threeSum([0])); // Output: []
```

### Explanation:

1. **Objective:** Find all unique triplets in the array that sum to zero.
2. **Approach:** Use a combination of sorting and the two-pointer technique:
  - o Fix one element (`nums[i]`) and use two pointers to find the other two elements (`nums[left]` and `nums[right]`) that sum to zero with `nums[i]`.
3. **Steps:**
  - o Sort the array to simplify duplicate removal and use the two-pointer technique.
  - o For each element:

- Skip duplicate values.
  - Use two pointers (left and right) to find pairs that sum to  $-\text{nums}[i]$ .
  - Push valid triplets to the result array and skip duplicates.
- Why it Works:** Sorting ensures that duplicates are handled efficiently and the two-pointer technique finds pairs in  $O(n)$  time.
  - Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
  - Space Complexity:**  $O(1)$  — No extra space, aside from the result array.

### Multi-Sum Combinations in Arrays (Closest Sum)

**72. Problem:** Given an integer array `nums` and an integer `target`, find the sum of the three integers such that the sum is closest to target. Return the sum. Assume that there will always be exactly one solution.

**Example:**

```
Input: nums = [-1, 2, 1, -4], target = 1
Output: 2
Explanation: The sum that is closest to 1 is 2 (-1 + 2 + 1 = 2).
```

**Solution:**

```
function threeSumClosest(nums, target) {
  nums.sort((a, b) => a - b); // Sort the array
  let closestSum = Infinity;

  for (let i = 0; i < nums.length - 2; i++) {
    let left = i + 1;
    let right = nums.length - 1;

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

      // Update closestSum if the current sum is closer to the target
      if (Math.abs(currentSum - target) < Math.abs(closestSum - target)) {
        closestSum = currentSum;
      }

      if (currentSum < target) {
        left++; // Increase the sum
      } else if (currentSum > target) {
        right--; // Decrease the sum
      } else {
        return currentSum; // Exact match
      }
    }
  }

  return closestSum;
}
```

```

}

// Test
console.log(threeSumClosest([-1, 2, 1, -4], 1)); // Output: 2
console.log(threeSumClosest([0, 0, 0], 1)); // Output: 0

```

### Explanation:

1. **Objective:** Find the sum of three numbers that is closest to the given target.
2. **Approach:**
  - o Sort the array to use the two-pointer technique.
  - o Fix one number (`nums[i]`) and adjust the two pointers (left and right) to find a sum close to the target.
3. **Steps:**
  - o Sort the array to simplify logic and handle duplicates.
  - o For each fixed number:
    - Calculate the current sum of the fixed number and the two-pointer values.
    - Compare the absolute difference between the current sum and the target with the closest sum found so far.
  - o Update `closestSum` whenever a closer sum is found.
4. **Why it Works:** Sorting and the two-pointer technique ensure efficient traversal and comparison for finding the closest sum.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(1)$  — No extra space, aside from the result.

### Multi-Sum Combinations in Arrays (All Triplets Less Than Target)

#### 73. Problem: Find all unique triplets in the array such that the sum of the triplet is less than a given target.

##### Example:

```

Input: (nums = [-2, 0, 1, 3]), (target = 2);
Output: [
  [-2, 0, 1],
  [-2, 0, 3],
  [-2, 1, 3],
  [0, 1, 3],
];

```

##### Solution:

```

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

  for (let i = 0; i < nums.length - 2; i++) {
    let left = i + 1;
    let right = nums.length - 1;

    while (left < right) {
      const sum = nums[i] + nums[left] + nums[right];
      if (sum < target) {
        result.push([nums[i], nums[left], nums[right]]);
        left++;
      } else if (sum > target) {
        right--;
      } else {
        result.push([nums[i], nums[left], nums[right]]);
        left++;
        right--;
      }
    }
  }
}

```

```

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

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

    if (sum < target) {
        for (let k = right; k > left; k--) {
            result.push([nums[i], nums[left], nums[k]]);
        }
        left++; // Move the left pointer
    } else {
        right--; // Move the right pointer
    }
}

return result;
}

// Test
console.log(threeSumLessThanTarget([-2, 0, 1, 3], 2)); // Output: [[-2, 0, 1], [-2, 0, 3], [-2, 1, 3], [0, 1, 3]]
console.log(threeSumLessThanTarget([1, 2, 3], 5)); // Output: [[1, 2, 3]]

```

### **Explanation:**

1. **Objective:** Find all unique triplets where the sum is less than the given target.
2. **Approach:**
  - o Sort the array to simplify logic.
  - o Fix one number and use two pointers to find pairs that satisfy the condition.
3. **Steps:**
  - o Sort the array to enable efficient traversal.
  - o For each fixed number:
    - Use the two-pointer technique to find all pairs where the sum of the triplet is less than the target.
    - Push the triplets into the result array.
4. **Why it Works:** Sorting ensures duplicate handling, and the two-pointer technique reduces unnecessary comparisons.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(n)$  — Space for storing the result array.

### **Multi-Sum Combinations in Arrays (Product Equal to Target)**

**74. Problem:** **Find all unique triplets in the array such that the product of the triplet equals a given target.**

### **Example:**

Input: (nums = [-1, -1, 1, 2, 3]), (target = 6);

Output: [  
[-1, -1, 6],  
[1, 2, 3],  
];

### Solution:

```
function threeSumProduct(nums, target) {  
    nums.sort((a, b) => a - b); // Sort the array  
    const result = [];  
  
    for (let i = 0; i < nums.length - 2; i++) {  
        if (i > 0 && nums[i] === nums[i - 1]) continue; // Skip duplicates  
  
        let left = i + 1;  
        let right = nums.length - 1;  
  
        while (left < right) {  
            const product = nums[i] * nums[left] * nums[right];  
  
            if (product === target) {  
                result.push([nums[i], nums[left], nums[right]]);  
  
                // Move pointers to skip duplicates  
                while (left < right && nums[left] === nums[left + 1]) left++;  
                while (left < right && nums[right] === nums[right - 1]) right--;  
  
                left++;  
                right--;  
            } else if (product < target) {  
                left++; // Increase product  
            } else {  
                right--; // Decrease product  
            }  
        }  
    }  
  
    return result;  
}  
  
// Test  
console.log(threeSumProduct([-1, -1, 1, 2, 3], 6)); // Output: [[-1, -1, 6], [1, 2, 3]]  
console.log(threeSumProduct([1, 2, 3, 4], 24)); // Output: [[1, 2, 3]]
```

### Explanation:

1. **Objective:** Find all triplets whose product equals the given target.
2. **Approach:**
  - o Use sorting and the two-pointer technique to efficiently find the triplets.
  - o Fix one number and adjust the two pointers based on the product.
3. **Steps:**
  - o Fix one number, then find pairs of numbers whose product matches the required value to achieve the target.
  - o Handle duplicates by skipping over repeated values.
4. **Why it Works:** Sorting ensures efficient traversal and avoids unnecessary computations.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(n)$  — Space for the result array.

### **Multi-Sum Combinations in Arrays (Smallest Sum Greater Than Target)**

**75. Problem: Find the smallest sum of three integers in the array that is strictly greater than a given target.**

**Example:**

Input: nums = [-1, 2, 1, -4], target = 1  
 Output: 2  
 Explanation: The smallest sum greater than 1 is 2 (-1 + 2 + 1 = 2).

Input: nums = [1, 2, 3, 4], target = 6  
 Output: 7  
 Explanation: The smallest sum greater than 6 is 7 (1 + 2 + 4 = 7).

**Solution:**

```
function threeSumSmallestGreater(nums, target) {
  nums.sort((a, b) => a - b); // Sort the array
  let smallestGreater = Infinity;

  for (let i = 0; i < nums.length - 2; i++) {
    let left = i + 1;
    let right = nums.length - 1;

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

      if (sum > target) {
        smallestGreater = Math.min(smallestGreater, sum);
        right--; // Decrease sum to get closer to target
      } else {
        left++; // Increase sum
      }
    }
  }
}
```

```

    }

    return smallestGreater === Infinity ? -1 : smallestGreater;
}

// Test
console.log(threeSumSmallestGreater([-1, 2, 1, -4], 1)); // Output: 2
console.log(threeSumSmallestGreater([1, 2, 3, 4], 6)); // Output: 7

```

### Explanation:

1. **Objective:** Find the smallest sum of three numbers that is strictly greater than the target.
2. **Approach:**
  - o Sort the array to simplify logic.
  - o Use a two-pointer approach to find triplets with sums greater than the target.
3. **Steps:**
  - o Fix one number (nums[i]) and adjust two pointers (left and right) to find sums greater than the target.
  - o Track the smallest sum greater than the target and update it when a valid sum is found.
  - o Return -1 if no valid sum is found.
4. **Why it Works:** Sorting allows efficient traversal and comparison using two pointers.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(1)$  — No extra space, aside from the result.

### Multi-Sum Combinations in Arrays (Pairs Within Range)

**76. Problem: Find all unique triplets in the array such that the sum of the triplet falls within a given range [low, high].**

#### Example:

```

Input: (nums = [-2, 0, 1, 3]), (low = 1), (high = 3);
Output: [
  [-2, 0, 3],
  [-2, 1, 3],
  [0, 1, 3],
];

```

#### Solution:

```

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

  for (let i = 0; i < nums.length - 2; i++) {

```

```

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

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

    if (sum >= low && sum <= high) {
        result.push([nums[i], nums[left], nums[right]]);
        left++;
        right--;
    } else if (sum < low) {
        left++; // Increase sum
    } else {
        right--; // Decrease sum
    }
}

return result;
}

// Test
console.log(threeSumInRange([-2, 0, 1, 3], 1, 3)); // Output: [[-2, 0, 3], [-2, 1, 3], [0, 1, 3]]
console.log(threeSumInRange([1, 2, 3, 4], 5, 10)); // Output: [[1, 2, 3], [1, 3, 4], [2, 3, 4]]

```

### **Explanation:**

1. **Objective:** Find all triplets whose sums fall within a given range.
2. **Approach:**
  - o Sort the array to enable efficient traversal with two pointers.
  - o Use the two-pointer technique to calculate sums and check if they fall within the range [low, high].
3. **Steps:**
  - o Fix one number (nums[i]) and adjust two pointers (left and right) to find pairs that satisfy the range condition.
  - o Push valid triplets into the result array.
4. **Why it Works:** Sorting simplifies duplicate handling and allows efficient use of two pointers to find valid triplets.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(n)$  — Space for the result array.

### **Multi-Sum Combinations in Arrays (With Multiset Inputs)**

**77. Problem:** Given a multiset (array with duplicates allowed), find all unique triplets that sum to zero.

### **Example:**

```

Input: (nums = [-1, -1, -1, 0, 1, 1, 2]), (target = 0);
Output: [
  [-1, -1, 2],
  [-1, 0, 1],
];

```

### Solution:

```

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

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

    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
        while (left < right && nums[left] === nums[left + 1]) left++;
        while (left < right && nums[right] === nums[right - 1]) right--;

        left++;
        right--;
      } else if (sum < 0) {
        left++; // Increase sum
      } else {
        right--; // Decrease sum
      }
    }
  }

  return result;
}

// Test
console.log(threeSumMultiset([-1, -1, -1, 0, 1, 1, 2])); // Output: [[-1, -1, 2], [-1, 0, 1]]
console.log(threeSumMultiset([0, 0, 0, 0])); // Output: [[0, 0, 0]]

```

### Explanation:

1. **Objective:** Handle duplicate values in the input array and return unique triplets.
2. **Approach:**
  - o Sort the array to simplify duplicate handling.
  - o Skip duplicates for the fixed element (`nums[i]`) and the two pointers (`nums[left]`, `nums[right]`).
3. **Steps:**
  - o Fix one number and use two pointers to find pairs that sum to zero with the fixed number.
  - o Skip over duplicate values to ensure triplets are unique.
4. **Why it Works:** Sorting ensures duplicates are easily skipped, and the two-pointer technique efficiently finds valid pairs.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(n)$  — Space for the result array.

### **Multi-Sum Combinations in Arrays (With Exact Count of Triplets)**

**78. Problem: Find all unique triplets in the array such that the sum of the triplet equals a given target and the triplets appear exactly once, even if the input contains duplicates.**

**Example:**

```
Input: (nums = [-1, -1, 2, 0, 1, -1, 1]), (target = 0);
Output: [
  [-1, -1, 2],
  [-1, 0, 1],
];
```

**Solution:**

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

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

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

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

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

        // Skip duplicates for left and right pointers
        while (left < right && nums[left] === nums[left + 1]) left++;
        while (left < right && nums[right] === nums[right - 1]) right--;
      }
    }
  }
}
```

```

        left++;
        right--;
    } else if (sum < target) {
        left++; // Increase sum
    } else {
        right--; // Decrease sum
    }
}

return result;
}

// Test
console.log(threeSumExactCount([-1, -1, 2, 0, 1, -1, 1], 0)); // Output: [[-1, -1, 2], [-1, 0, 1]]
console.log(threeSumExactCount([0, 0, 0, 0], 0)); // Output: [[0, 0, 0]]

```

### Explanation:

1. **Objective:** Return all unique triplets where the sum equals the target, ensuring duplicates are ignored.
2. **Approach:**
  - o Sort the array to simplify duplicate handling.
  - o Skip duplicates for the fixed element and both pointers to ensure unique triplets.
3. **Steps:**
  - o Fix one number and use two pointers to find pairs that sum to the target with the fixed number.
  - o Skip over duplicates for both the fixed number and the pointers.
4. **Why it Works:** Sorting allows efficient duplicate handling and ensures no repeated triplets in the result.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(n)$  — Space for the result array.

### Multi-Sum Combinations in Arrays (Triplets with Maximum Product)

#### 79. Problem: Find the triplet in the array that has the maximum product.

##### Example:

Input: `nums = [-10, -10, 5, 2]`  
Output: `[-10, -10, 5]`  
Explanation: The maximum product is 500 ( $-10 \times -10 \times 5 = 500$ ).

##### Solution:

```

function threeSumMaxProduct(nums) {
    nums.sort((a, b) => a - b); // Sort the array

    // The max product can either be:
    // 1. The product of the three largest numbers
    // 2. The product of the two smallest numbers (negative) and the largest number
    const n = nums.length;
    const product1 = nums[n - 1] * nums[n - 2] * nums[n - 3];
    const product2 = nums[0] * nums[1] * nums[n - 1];

    return product1 > product2
        ? [nums[n - 3], nums[n - 2], nums[n - 1]]
        : [nums[0], nums[1], nums[n - 1]];
}

// Test
console.log(threeSumMaxProduct([-10, -10, 5, 2])); // Output: [-10, -10, 5]
console.log(threeSumMaxProduct([1, 2, 3])); // Output: [1, 2, 3]

```

### Explanation:

1. **Objective:** Find the triplet with the largest product.
2. **Approach:**
  - o Sort the array to simplify finding the largest and smallest numbers.
  - o Compare the product of the three largest numbers with the product of the two smallest (negative) numbers and the largest number.
3. **Steps:**
  - o Sort the array.
  - o Compute the two possible maximum products and return the triplet with the larger product.
4. **Why it Works:** Sorting ensures efficient access to the largest and smallest values in the array.
5. **Time Complexity:**  $O(n \log n)$  — Sorting the array.
6. **Space Complexity:**  $O(1)$  — No extra space, aside from the result.

### Multi-Sum Combinations in Arrays (All Triplets Divisible by k)

**80. Problem:** **Find all unique triplets in the array such that the sum of the triplet is divisible by k.**

#### Example:

Input: (nums = [3, 6, 9, 12]), (k = 6);  
Output: [[3, 6, 9]];

#### Solution:

```

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

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

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

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

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

                // Skip duplicates for left and right pointers
                while (left < right && nums[left] === nums[left + 1]) left++;
                while (left < right && nums[right] === nums[right - 1]) right--;

                left++;
                right--;
            } else if (sum < 0) {
                left++; // Increase sum
            } else {
                right--; // Decrease sum
            }
        }
    }

    return result;
}

// Test
console.log(threeSumDivisibleByK([3, 6, 9, 12], 6)); // Output: [[3, 6, 9]]
console.log(threeSumDivisibleByK([1, 2, 3, 4, 5], 3)); // Output: []

```

### Explanation:

1. **Objective:** Find all unique triplets where the sum is divisible by k.
2. **Approach:**
  - o Sort the array to simplify logic and handle duplicates.
  - o Use the two-pointer technique to find pairs that satisfy the divisibility condition with the fixed number.
3. **Steps:**
  - o Fix one number and use two pointers to find valid pairs.
  - o Skip duplicates for both the fixed number and the pointers.

4. **Why it Works:** Sorting enables efficient traversal and duplicate handling, while the modulus operation ensures divisibility.
5. **Time Complexity:**  $O(n^2)$  — Outer loop + two-pointer traversal.
6. **Space Complexity:**  $O(n)$  — Space for the result array.

## Array Product Computation

### Array Product Computation (Basic)

**81. Problem:** Given an integer array `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`. Solve it without using division and in  $O(n)$  time complexity.

**Example:**

```
Input: nums = [1, 2, 3, 4]
Output: [24, 12, 8, 6]
Explanation:
- For index 0: Product = 2 × 3 × 4 = 24
- For index 1: Product = 1 × 3 × 4 = 12
- For index 2: Product = 1 × 2 × 4 = 8
- For index 3: Product = 1 × 2 × 3 = 6
```

**Solution:**

```
function productExceptSelf(nums) {
  const n = nums.length;
  const output = Array(n).fill(1);

  let prefix = 1;
  for (let i = 0; i < n; i++) {
    output[i] = prefix;
    prefix *= nums[i];
  }

  let postfix = 1;
  for (let i = n - 1; i >= 0; i--) {
    output[i] *= postfix;
    postfix *= nums[i];
  }

  return output;
}
```

```
// Test
console.log(productExceptSelf([1, 2, 3, 4])); // Output: [24, 12, 8, 6]
console.log(productExceptSelf([0, 1])); // Output: [1, 0]
```

### Explanation:

1. **Objective:** Compute the product of all elements except the current one without using division.
2. **Approach:**
  - o Use two passes:
    - First pass (prefix): Compute the product of all elements before each index.
    - Second pass (postfix): Compute the product of all elements after each index and combine with the prefix product.
3. **Steps:**
  - o Initialize an output array filled with 1.
  - o Traverse the array from left to right to calculate prefix products.
  - o Traverse the array from right to left to calculate postfix products and update the output.
4. **Why it Works:** By combining prefix and postfix products, each index gets the product of all elements except itself.
5. **Time Complexity:**  $O(n)$  — Two passes through the array.
6. **Space Complexity:**  $O(1)$  — The output array does not count as extra space.

### Array Product Computation (With Division)

**82. Problem:** Given an integer array `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`. This time, solve it using the division method.

### Example:

```
Input: nums = [1, 2, 3, 4];
Output: [24, 12, 8, 6];
```

### Solution:

```
function productExceptSelfWithDivision(nums) {
  const totalProduct = nums.reduce((product, num) => product * num, 1);
  const zeroCount = nums.filter((num) => num === 0).length;

  if (zeroCount > 1) return Array(nums.length).fill(0); // More than one zero, all products are zero

  return nums.map((num) => {
    if (zeroCount === 1) return num === 0 ? totalProduct : 0;
    return totalProduct / num;
  });
}
```

```

});  

}  
  

// Test  

console.log(productExceptSelfWithDivision([1, 2, 3, 4])); // Output: [24, 12, 8, 6]  

console.log(productExceptSelfWithDivision([0, 1, 2])); // Output: [2, 0, 0]  

console.log(productExceptSelfWithDivision([0, 0, 2])); // Output: [0, 0, 0]

```

### **Explanation:**

1. **Objective:** Solve the problem using division.
2. **Approach:**
  - o Calculate the total product of all elements.
  - o Handle special cases with zeroes:
    - If there is more than one zero, all products are zero.
    - If there is exactly one zero, only the position of the zero will have a non-zero product.
  - o Use the total product to compute the result for each index.
3. **Steps:**
  - o Compute the total product using reduce.
  - o Check the number of zeroes in the array:
    - If zeroCount > 1, return an array filled with zeroes.
    - If zeroCount === 1, return the product only for the zero index.
    - Otherwise, divide the total product by each element.
4. **Why it Works:** Division provides an efficient way to calculate the product except self, with adjustments for zeroes.
5. **Time Complexity:** O(n) — Single pass to calculate the total product and another to map results.
6. **Space Complexity:** O(1) — No extra space, aside from the output array.

### **Array Product Computation (With Left and Right Arrays)**

#### **83. Problem: Solve the problem using separate arrays for prefix and suffix products.**

##### **Example:**

```

Input: nums = [1, 2, 3, 4];  

Output: [24, 12, 8, 6];

```

##### **Solution:**

```

function productExceptSelfWithArrays(nums) {  

  const n = nums.length;  

  const left = Array(n).fill(1);  

  const right = Array(n).fill(1);  

  const output = Array(n);
}

```

```

for (let i = 1; i < n; i++) {
    left[i] = left[i - 1] * nums[i - 1]; // Build left products
}

for (let i = n - 2; i >= 0; i--) {
    right[i] = right[i + 1] * nums[i + 1]; // Build right products
}

for (let i = 0; i < n; i++) {
    output[i] = left[i] * right[i]; // Combine left and right products
}

return output;
}

// Test
console.log(productExceptSelfWithArrays([1, 2, 3, 4])); // Output: [24, 12, 8, 6]
console.log(productExceptSelfWithArrays([5, 6, 2])); // Output: [12, 10, 30]

```

### **Explanation:**

1. **Objective:** Use prefix and suffix arrays to compute the result.
2. **Approach:**
  - o Create two arrays:
    - left[i]: Product of all elements to the left of index i.
    - right[i]: Product of all elements to the right of index i.
  - o Combine these arrays to compute the result.
3. **Steps:**
  - o Compute left by multiplying elements to the left of the current index.
  - o Compute right by multiplying elements to the right of the current index.
  - o Multiply left[i] and right[i] to get the result for index i.
4. **Why it Works:** The two arrays store all necessary information for computing the product without using division.
5. **Time Complexity:** O(n) — Three passes through the array.
6. **Space Complexity:** O(n) — Space for the left and right arrays.

### **Array Product Computation (Handling Large Numbers)**

#### **84. Problem: Modify the solution to avoid overflow when working with very large numbers.**

##### **Example:**

Input: nums = [1e9, 1e9, 1e9];  
 Output: [1e18, 1e18, 1e18];

##### **Solution:**

```

function productExceptSelfWithModulo(nums, mod = 1e9 + 7) {
  const n = nums.length;
  const output = Array(n).fill(1);

  let prefix = 1;
  for (let i = 0; i < n; i++) {
    output[i] = prefix;
    prefix = (prefix * nums[i]) % mod; // Use modulo to prevent overflow
  }

  let postfix = 1;
  for (let i = n - 1; i >= 0; i--) {
    output[i] = (output[i] * postfix) % mod; // Use modulo for each calculation
    postfix = (postfix * nums[i]) % mod;
  }

  return output;
}

// Test
console.log(productExceptSelfWithModulo([1e9, 1e9, 1e9])); // Output: [1e18 % 1e9+7, 1e18 % 1e9+7, 1e18 % 1e9+7]
console.log(productExceptSelfWithModulo([2, 3, 4, 5])); // Output: [60, 40, 30, 24]

```

### **Explanation:**

1. **Objective:** Avoid overflow when calculating products with large numbers.
2. **Approach:**
  - o Use a modulo operation to keep intermediate products manageable.
  - o Combine prefix and postfix products as before, applying modulo at each step.
3. **Steps:**
  - o Compute prefix and postfix products as in the optimized solution.
  - o Apply modulo operation to intermediate results to prevent overflow.
4. **Why it Works:** Modulo arithmetic ensures numbers remain within a manageable range while maintaining correctness.
5. **Time Complexity:** O(n) — Two passes through the array.
6. **Space Complexity:** O(1) — No extra space, aside from the output array.

### **Array Product Computation (With Negative Numbers)**

**85. Problem:** Handle cases where the input array contains negative numbers, ensuring the product is correctly calculated.

#### **Example:**

Input: `nums = [-1, 2, -3, 4]`

Output: `[-24, 12, -8, 6]`

Explanation:

- For index 0: Product =  $2 \times -3 \times 4 = -24$
- For index 1: Product =  $-1 \times -3 \times 4 = 12$
- For index 2: Product =  $-1 \times 2 \times 4 = -8$
- For index 3: Product =  $-1 \times 2 \times -3 = 6$

### Solution:

```
function productExceptSelfWithNegatives(nums) {
  const n = nums.length;
  const output = Array(n).fill(1);

  let prefix = 1;
  for (let i = 0; i < n; i++) {
    output[i] = prefix;
    prefix *= nums[i];
  }

  let postfix = 1;
  for (let i = n - 1; i >= 0; i--) {
    output[i] *= postfix;
    postfix *= nums[i];
  }

  return output;
}

// Test
console.log(productExceptSelfWithNegatives([-1, 2, -3, 4])); // Output: [-24, 12, -8, 6]
console.log(productExceptSelfWithNegatives([-1, -1, 1, 1])); // Output: [1, 1, -1, -1]
```

### Explanation:

- Objective:** Compute the product of all elements except the current one, accounting for negative numbers.
- Approach:**
  - Use the same prefix and postfix technique as in the basic solution.
  - Negative numbers are handled naturally in multiplication.
- Steps:**
  - Calculate prefix and postfix products as usual.
  - Combine them to get the result for each index.
- Why it Works:** The prefix-postfix method handles signs automatically because multiplication is commutative.
- Time Complexity:** O(n) — Two passes through the array.
- Space Complexity:** O(1) — No extra space, aside from the output array.

### Array Product Computation (With Zero at Multiple Indices)

## 86. Problem: Handle cases where the input array contains multiple zeroes, ensuring the output is correctly calculated.

### Example:

```
Input: nums = [0, 2, 0, 4];
Output: [0, 0, 0, 0];
```

```
Input: nums = [0, 2, 3, 4];
Output: [24, 0, 0, 0];
```

### Solution:

```
function productExceptSelfWithZeroes(nums) {
  const n = nums.length;
  const output = Array(n).fill(0);
  const totalProduct = nums.reduce(
    (prod, num) => (num !== 0 ? prod * num : prod),
    1
  );
  const zeroCount = nums.filter((num) => num === 0).length;

  if (zeroCount > 1) return output; // All zeros if more than one zero

  for (let i = 0; i < n; i++) {
    if (nums[i] === 0) {
      output[i] = totalProduct; // Product at the zero index
    } else if (zeroCount === 0) {
      output[i] = totalProduct / nums[i]; // Regular division
    }
  }

  return output;
}

// Test
console.log(productExceptSelfWithZeroes([0, 2, 0, 4])); // Output: [0, 0, 0, 0]
console.log(productExceptSelfWithZeroes([0, 2, 3, 4])); // Output: [24, 0, 0, 0]
```

### Explanation:

1. **Objective:** Handle arrays with zeroes to ensure correctness.
2. **Approach:**
  - o Compute the total product of all non-zero elements.
  - o Count the number of zeroes in the array:
    - If more than one zero, all outputs are zero.

- If exactly one zero, set the product at the zero index to the total product.
- Steps:**
    - Use reduce to calculate the product of non-zero elements.
    - Iterate through the array to handle cases with zero.
  - Why it Works:** By separately handling the zero cases, we ensure correctness without unnecessary calculations.
  - Time Complexity:**  $O(n)$  — Single pass to calculate the product and another to fill the output.
  - Space Complexity:**  $O(1)$  — No extra space, aside from the output array.

### Array Product Computation (Using Functional Programming)

**87. Problem: Solve the problem using functional programming methods like map, reduce, and no explicit loops.**

**Example:**

```
Input: nums = [1, 2, 3, 4];
Output: [24, 12, 8, 6];
```

**Solution:**

```
function productExceptSelfFunctional(nums) {
  const prefix = nums.reduce((acc, num, i) => {
    acc.push((acc[i - 1] || 1) * num);
    return acc;
  }, []);
}

let postfix = 1;
return nums.map((num, i) => {
  const result = (prefix[i - 1] || 1) * postfix;
  postfix *= num;
  return result;
});
}

// Test
console.log(productExceptSelfFunctional([1, 2, 3, 4])); // Output: [24, 12, 8, 6]
console.log(productExceptSelfFunctional([2, 3, 4])); // Output: [12, 8, 6]
```

**Explanation:**

- Objective:** Use functional programming constructs to solve the problem.
- Approach:**
  - Use reduce to compute the prefix products.
  - Use map to calculate the output by combining prefix and postfix products.

3. **Steps:**
  - o Compute the prefix array with reduce.
  - o Use a single pass with map to calculate the result, maintaining the postfix product as a variable.
4. **Why it Works:** Functional constructs allow a concise and readable implementation without explicit loops.
5. **Time Complexity:**  $O(n)$  — Single pass for prefix calculation and another for mapping the result.
6. **Space Complexity:**  $O(n)$  — Space for the prefix array.

### **Array Product Computation (With Modulo for Overflow Handling)**

**88. Problem: Modify the solution to handle overflow for very large products by using modulo arithmetic.**

**Example:**

```
Input: (nums = [1e9, 1e9, 1e9, 2]), (mod = 1e9 + 7);
Output: [2000000000, 2000000000, 2000000000, 1000000000] % (1e9 + 7);
```

**Solution:**

```
function productExceptSelfWithModulo(nums, mod = 1e9 + 7) {
  const n = nums.length;
  const output = Array(n).fill(1);

  let prefix = 1;
  for (let i = 0; i < n; i++) {
    output[i] = prefix;
    prefix = (prefix * nums[i]) % mod; // Use modulo to prevent overflow
  }

  let postfix = 1;
  for (let i = n - 1; i >= 0; i--) {
    output[i] = (output[i] * postfix) % mod; // Use modulo to keep results manageable
    postfix = (postfix * nums[i]) % mod;
  }

  return output;
}

// Test
console.log(productExceptSelfWithModulo([1e9, 1e9, 1e9, 2])); // Output: [2000000000, 2000000000, 2000000000, 1000000000] % (1e9 + 7)
console.log(productExceptSelfWithModulo([2, 3, 4, 5])); // Output: [60, 40, 30, 24]
```

**Explanation:**

1. **Objective:** Handle very large numbers using modulo arithmetic to prevent overflow.
2. **Approach:**
  - o Use prefix and postfix techniques but apply modulo operations during multiplications.
3. **Steps:**
  - o Compute the prefix product modulo mod for each element.
  - o Compute the postfix product modulo mod for each element and combine it with the prefix product.
4. **Why it Works:** Modulo arithmetic ensures all intermediate results stay within manageable limits without affecting correctness.
5. **Time Complexity:**  $O(n)$  — Two passes through the array.
6. **Space Complexity:**  $O(1)$  — No extra space, aside from the output array.

## 89. Array Product Computation (With Sparse Arrays)

### Problem:

Handle sparse arrays (arrays with many zeroes) efficiently, ensuring correctness without unnecessary calculations.

### Example:

```
Input: nums = [0, 0, 2, 0, 3];
Output: [0, 0, 0, 0, 0];
```

```
Input: nums = [0, 2, 3, 4, 0];
Output: [0, 0, 0, 0, 0];
```

### Solution:

```
function productExceptSelfSparse(nums) {
  const totalProduct = nums.reduce(
    (prod, num) => (num === 0 ? prod : prod * num),
    1
  );
  const zeroCount = nums.filter((num) => num === 0).length;

  return nums.map((num) => {
    if (zeroCount > 1) return 0; // If more than one zero, all outputs are zero
    if (zeroCount === 1) return num === 0 ? totalProduct : 0; // Only one zero
    return totalProduct / num; // No zeroes
  });
}

// Test
console.log(productExceptSelfSparse([0, 0, 2, 0, 3])); // Output: [0, 0, 0, 0, 0]
console.log(productExceptSelfSparse([0, 2, 3, 4, 0])); // Output: [0, 0, 0, 0, 0]
console.log(productExceptSelfSparse([1, 2, 3])); // Output: [6, 3, 2]
```

### **Explanation:**

1. **Objective:** Handle sparse arrays efficiently, minimizing unnecessary computations for zeroes.
2. **Approach:**
  - o Calculate the total product of non-zero elements.
  - o Check the number of zeroes:
    - If zeroCount > 1, all outputs are zero.
    - If zeroCount === 1, only the position of the zero has a non-zero product.
3. **Steps:**
  - o Use reduce to compute the total product for non-zero elements.
  - o Iterate through the array to assign values based on zero counts.
4. **Why it Works:** Explicitly handling zeroes avoids redundant calculations and ensures correctness.
5. **Time Complexity:**  $O(n)$  — Single pass to calculate the total product and another to map results.
6. **Space Complexity:**  $O(1)$  — No extra space, aside from the output array.

### **Array Product Computation (Functional and Optimized)**

**90. Problem: Implement the solution using functional programming constructs like map and reduce in a concise and optimized manner.**

#### **Example:**

```
Input: nums = [1, 2, 3, 4];
Output: [24, 12, 8, 6];
```

#### **Solution:**

```
function productExceptSelfFunctionalOptimized(nums) {
  const n = nums.length;

  // Build prefix products
  const prefix = nums.reduce((acc, num, i) => {
    acc.push((acc[i - 1] || 1) * num);
    return acc;
  }, []);

  let postfix = 1;
  return nums.map(_, i) => {
    const result = (prefix[i - 1] || 1) * postfix;
    postfix *= nums[i];
    return result;
  };
}

// Test
```

```
console.log(productExceptSelfFunctionalOptimized([1, 2, 3, 4])); // Output: [24, 12, 8, 6]
console.log(productExceptSelfFunctionalOptimized([5, 6, 2])); // Output: [12, 10, 30]
```

### Explanation:

1. **Objective:** Use functional programming constructs to implement the solution in a concise and optimized way.
2. **Approach:**
  - o Compute the prefix products using reduce.
  - o Compute the result by combining prefix and postfix products with a map.
3. **Steps:**
  - o Use reduce to compute an array of prefix products.
  - o Use map to calculate the final output by maintaining a running postfix product.
4. **Why it Works:** Functional programming ensures clear, readable, and concise code, reducing the need for explicit loops.
5. **Time Complexity:**  $O(n)$  — Single pass for prefix calculation and another for mapping the result.
6. **Space Complexity:**  $O(n)$  — Space for the prefix array.

## Water Collection in Structures :

### Water Collection in Structures (Two Pointer Approach)

**91. Problem:** Given an array height of non-negative integers, where each element represents the height of a vertical line, find two lines that together with the x-axis form a container that can hold the most water. Return the maximum amount of water the container can store.

### Example:

Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7]

Output: 49

Explanation: The two lines at index 1 and 8 (height 8 and 7) form a container with an area of 49.

### Solution:

```
function maxArea(height) {
  let left = 0;
  let right = height.length - 1;
  let maxWater = 0;

  while (left < right) {
    const width = right - left;
    const currentHeight = Math.min(height[left], height[right]);
```

```

const currentArea = width * currentHeight;

maxWater = Math.max(maxWater, currentArea);

// Move the smaller height inward to try for a larger area
if (height[left] < height[right]) {
    left++;
} else {
    right--;
}
}

return maxWater;
}

// Test
console.log(maxArea([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: 49
console.log(maxArea([1, 1])); // Output: 1
console.log(maxArea([4, 3, 2, 1, 4])); // Output: 16

```

### **Explanation:**

1. **Objective:** Find the maximum area that can be formed between two vertical lines.
2. **Approach:**
  - o Use the two-pointer technique:
    - Start with the left pointer at index 0 and the right pointer at the last index.
    - Calculate the area and update the maximum area found so far.
    - Move the pointer with the smaller height inward to try for a larger area.
3. **Steps:**
  - o Calculate the width as right - left.
  - o Use the smaller of the two heights as the height of the container.
  - o Update the maximum area and adjust the pointers.
4. **Why it Works:** By always moving the smaller height inward, we maximize the potential for a larger area.
5. **Time Complexity:** O(n) — Each pointer moves at most once for every element.
6. **Space Complexity:** O(1) — Uses only a few variables.

### **Water Collection in Structures (Brute Force Approach)**

**92. Problem:** **Find the maximum area of water a container can store using a brute force method.**

#### **Example:**

Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7];  
Output: 49;

```
Input: height = [1, 1];
Output: 1;
```

### Solution:

```
function maxAreaBruteForce(height) {
    let maxWater = 0;

    for (let i = 0; i < height.length; i++) {
        for (let j = i + 1; j < height.length; j++) {
            const width = j - i;
            const currentHeight = Math.min(height[i], height[j]);
            const currentArea = width * currentHeight;

            maxWater = Math.max(maxWater, currentArea);
        }
    }

    return maxWater;
}

// Test
console.log(maxAreaBruteForce([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: 49
console.log(maxAreaBruteForce([1, 1])); // Output: 1
console.log(maxAreaBruteForce([4, 3, 2, 1, 4])); // Output: 16
```

### Explanation:

1. **Objective:** Calculate the maximum area by comparing all possible pairs of lines.
2. **Approach:**
  - o Use two nested loops to calculate the area for every possible pair of lines.
  - o For each pair, calculate the width ( $j - i$ ) and height ( $\min(\text{height}[i], \text{height}[j])$ ) of the container.
  - o Update the maximum area found so far.
3. **Why it Works:** The brute force method explores all combinations, ensuring correctness.
4. **Time Complexity:**  $O(n^2)$  — For every element, compare it with all subsequent elements.
5. **Space Complexity:**  $O(1)$  — Uses only a few variables.

### Water Collection in Structures (Track Indices of the Lines)

**93. Problem:** **Return the indices of the two lines that form the container with the most water.**

### Example:

```
Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
```

Output: [1, 8]

Explanation: The lines at index 1 and 8 form the container with the largest area of 49.

### Solution:

```
function maxAreaWithIndices(height) {  
    let left = 0;  
    let right = height.length - 1;  
    let maxWater = 0;  
    let resultIndices = [0, 0];  
  
    while (left < right) {  
        const width = right - left;  
        const currentHeight = Math.min(height[left], height[right]);  
        const currentArea = width * currentHeight;  
  
        if (currentArea > maxWater) {  
            maxWater = currentArea;  
            resultIndices = [left, right];  
        }  
  
        if (height[left] < height[right]) {  
            left++;  
        } else {  
            right--;  
        }  
    }  
  
    return resultIndices;  
}  
  
// Test  
console.log(maxAreaWithIndices([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: [1, 8]  
console.log(maxAreaWithIndices([1, 1])); // Output: [0, 1]  
console.log(maxAreaWithIndices([4, 3, 2, 1, 4])); // Output: [0, 4]
```

### Explanation:

1. **Objective:** Find the indices of the two lines that form the container with the maximum area.
2. **Approach:**
  - o Use the two-pointer method to maximize the area.
  - o Track the indices of the lines whenever a new maximum area is found.
3. **Steps:**
  - o Calculate the width and height for the current left and right pointers.
  - o Update the maximum area and the indices if a new maximum is found.
  - o Move the pointer with the smaller height inward to try for a larger area.

4. **Why it Works:** Tracking indices alongside the area calculation provides the correct lines for the maximum container.
5. **Time Complexity:**  $O(n)$  — Each pointer moves at most once for every element.
6. **Space Complexity:**  $O(1)$  — Uses only a few variables.

### Water Collection in Structures (Handle Edge Cases)

**94. Problem: Handle edge cases, such as an array with fewer than two elements or all elements being the same.**

**Example:**

```
Input: height = [1]
Output: 0
```

```
Input: height = [4, 4, 4, 4]
Output: 12
```

Explanation: The largest container is formed between indices 0 and 3, with an area of  $4 \times 3 = 12$ .

**Solution:**

```
function maxAreaHandleEdgeCases(height) {
  if (height.length < 2) return 0; // Not enough lines to form a container

  let left = 0;
  let right = height.length - 1;
  let maxWater = 0;

  while (left < right) {
    const width = right - left;
    const currentHeight = Math.min(height[left], height[right]);
    const currentArea = width * currentHeight;

    maxWater = Math.max(maxWater, currentArea);

    if (height[left] < height[right]) {
      left++;
    } else {
      right--;
    }
  }

  return maxWater;
}

// Test
console.log(maxAreaHandleEdgeCases([1])); // Output: 0
```

```
console.log(maxAreaHandleEdgeCases([4, 4, 4])); // Output: 8  
console.log(maxAreaHandleEdgeCases([1, 2])); // Output: 1
```

### Explanation:

1. **Objective:** Handle edge cases gracefully, ensuring correctness even for small or special inputs.
2. **Approach:**
  - o Check if the input array has fewer than two elements. If so, return 0 because no container can be formed.
  - o Use the two-pointer approach for valid cases.
3. **Steps:**
  - o Handle arrays with fewer than two elements explicitly.
  - o Apply the standard two-pointer logic for all other cases.
4. **Why it Works:** Handling edge cases separately ensures correctness for all inputs.
5. **Time Complexity:**  $O(n)$  — Each pointer moves at most once for every element.
6. **Space Complexity:**  $O(1)$  — Uses only a few variables.

## Water Collection in Structures (Maximum Area in Rotated Input)

### 95. Problem: Handle the case where the input array is rotated, and find the maximum water container.

#### Example:

```
Input: height = [5, 4, 3, 8, 6, 2]  
Output: 15  
Explanation: The lines at index 0 and 3 (heights 5 and 8) form the largest container with an area of  $5 \times 3 = 15$ .
```

#### Solution:

```
function maxAreaRotatedInput(height) {  
    let left = 0;  
    let right = height.length - 1;  
    let maxWater = 0;  
  
    while (left < right) {  
        const width = right - left;  
        const currentHeight = Math.min(height[left], height[right]);  
        const currentArea = width * currentHeight;  
  
        maxWater = Math.max(maxWater, currentArea);  
  
        // Move pointers inward based on the heights  
        if (height[left] <= height[right]) {  
            left++;  
        } else {
```

```

        right--;
    }
}

return maxWater;
}

// Test
console.log(maxAreaRotatedInput([5, 4, 3, 8, 6, 2])); // Output: 15
console.log(maxAreaRotatedInput([7, 1, 2, 3, 9])); // Output: 28
console.log(maxAreaRotatedInput([1, 2, 4, 3])); // Output: 4

```

### Explanation:

1. **Objective:** Find the maximum container area for arrays where the heights are rotated or unsorted.
2. **Approach:**
  - o Use the two-pointer technique:
    - Start with pointers at both ends.
    - Move the pointer corresponding to the smaller height inward.
  - o Calculate the area at each step and update the maximum area.
3. **Steps:**
  - o Calculate the width and height at the current pointers.
  - o Update the maximum area if the current area is larger.
  - o Adjust pointers based on the smaller height.
4. **Why it Works:** The two-pointer method efficiently narrows down potential maximum areas regardless of the array's order.
5. **Time Complexity:** O(n) — Each pointer moves at most once for every element.
6. **Space Complexity:** O(1) — Uses only a few variables.

### Water Collection in Structures (Binary Search Optimization)

**96. Problem: Use binary search to optimize the search for the container with the most water.**

#### Example:

Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7];  
Output: 49;

#### Solution:

```

function maxAreaBinarySearch(height) {
    let left = 0;
    let right = height.length - 1;
    let maxWater = 0;

    while (left < right) {

```

```

const width = right - left;
const currentHeight = Math.min(height[left], height[right]);
const currentArea = width * currentHeight;

maxWater = Math.max(maxWater, currentArea);

// Move the smaller pointer inward
if (height[left] < height[right]) {
    let nextLeft = left + 1;
    while (nextLeft < right && height[nextLeft] <= height[left]) {
        nextLeft++; // Skip non-promising heights
    }
    left = nextLeft;
} else {
    let nextRight = right - 1;
    while (nextRight > left && height[nextRight] <= height[right]) {
        nextRight--; // Skip non-promising heights
    }
    right = nextRight;
}

return maxWater;
}

// Test
console.log(maxAreaBinarySearch([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: 49
console.log(maxAreaBinarySearch([4, 3, 2, 1, 4])); // Output: 16

```

### Explanation:

1. **Objective:** Optimize the two-pointer method further using binary search principles to skip non-promising indices.
2. **Approach:**
  - o Skip heights that are not larger than the previous height at the pointer.
  - o Move the pointers inward more aggressively when the next height is smaller.
3. **Steps:**
  - o Use the two-pointer approach with additional logic to skip indices that do not contribute to a larger area.
4. **Why it Works:** Skipping non-promising heights reduces unnecessary comparisons and iterations.
5. **Time Complexity:**  $O(n)$  — Skips redundant comparisons but still linear.
6. **Space Complexity:**  $O(1)$  — Uses only a few variables.

### Water Collection in Structures (Functional Programming)

**97. Problem: Implement the solution using functional programming constructs like reduce for a concise implementation.**

### Example:

```
Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7];
Output: 49;
```

### Solution:

```
function maxAreaFunctional(height) {
  return height.reduce((maxWater, _, left) => {
    for (let right = height.length - 1; right > left; right--) {
      const width = right - left;
      const currentHeight = Math.min(height[left], height[right]);
      maxWater = Math.max(maxWater, width * currentHeight);
    }
    return maxWater;
  }, 0);
}

// Test
console.log(maxAreaFunctional([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: 49
console.log(maxAreaFunctional([4, 3, 2, 1, 4])); // Output: 16
```

### Explanation:

1. **Objective:** Solve the problem using a functional approach.
2. **Approach:**
  - o Use reduce to iterate through all possible pairs.
  - o Calculate the area for each pair and update the maximum.
3. **Steps:**
  - o Iterate through all pairs using a nested loop within the reduce callback.
  - o Return the maximum area found.
4. **Why it Works:** Functional constructs provide a concise way to implement brute-force logic.
5. **Time Complexity:**  $O(n^2)$  — Nested loop in the functional approach.
6. **Space Complexity:**  $O(1)$  — No additional space used.

## Water Collection in Structures (Find Minimum Height for Maximum Area)

### 98. Problem: Find the minimum height of the two lines that form the container with the maximum area.

### Example:

```
Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
Output: 7
```

Explanation: The two lines with the maximum area are at indices 1 and 8, with heights 8 and 7. The minimum height is 7.

### Solution:

```
function minHeightForMaxArea(height) {  
    let left = 0;  
    let right = height.length - 1;  
    let maxWater = 0;  
    let minHeight = 0;  
  
    while (left < right) {  
        const width = right - left;  
        const currentHeight = Math.min(height[left], height[right]);  
        const currentArea = width * currentHeight;  
  
        if (currentArea > maxWater) {  
            maxWater = currentArea;  
            minHeight = currentHeight;  
        }  
  
        if (height[left] < height[right]) {  
            left++;  
        } else {  
            right--;  
        }  
    }  
  
    return minHeight;  
}  
  
// Test  
console.log(minHeightForMaxArea([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: 7  
console.log(minHeightForMaxArea([4, 3, 2, 1, 4])); // Output: 4  
console.log(minHeightForMaxArea([1, 1])); // Output: 1
```

### Explanation:

1. **Objective:** Determine the minimum height of the two lines that form the container with the maximum area.
2. **Approach:**
  - o Use the two-pointer technique to calculate the maximum area.
  - o Track the minimum height whenever a new maximum area is found.
3. **Steps:**
  - o Calculate the width and height at each step.
  - o Update the maximum area and record the minimum height for the lines forming it.
4. **Why it Works:** By storing the minimum height at the point of maximum area, we ensure correct results.
5. **Time Complexity:** O(n) — Each pointer moves at most once for every element.
6. **Space Complexity:** O(1) — Uses only a few variables.

## Water Collection in Structures (Area at Midpoints)

**99. Problem:** Calculate the maximum area formed by the two midpoints in the array.

**Example:**

Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7]

Output: 24

Explanation: The two midpoints are at indices 3 and 5, with heights 2 and 4. The area is  $3 \times 4 = 12$ .

**Solution:**

```
function maxAreaAtMidpoints(height) {  
    const midLeft = Math.floor((height.length - 1) / 2);  
    const midRight = Math.ceil((height.length - 1) / 2);  
  
    const width = midRight - midLeft;  
    const currentHeight = Math.min(height[midLeft], height[midRight]);  
  
    return width * currentHeight;  
}  
  
// Test  
console.log(maxAreaAtMidpoints([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: 4  
console.log(maxAreaAtMidpoints([4, 3, 2, 1, 4])); // Output: 3  
console.log(maxAreaAtMidpoints([1, 1])); // Output: 0
```

**Explanation:**

1. **Objective:** Calculate the container area using the two midpoint indices of the array.
2. **Approach:**
  - o Find the indices of the two midpoints of the array.
  - o Use these indices to calculate the area based on their heights and the distance between them.
3. **Steps:**
  - o Determine the midpoints using Math.floor and Math.ceil.
  - o Compute the area formed by these two lines.
4. **Why it Works:** Using midpoints provides a specific reference for calculating the area.
5. **Time Complexity:** O(1) — Constant time calculation.
6. **Space Complexity:** O(1) — No additional space used.

## Water Collection in Structures (Using Precomputed Prefix and Suffix Max Heights)

**100. Problem:** Precompute the maximum heights from both ends and use them to optimize the area calculation.

### Example:

```
Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7];
Output: 49;
```

```
function maxAreaWithPrecomputation(height) {
    const n = height.length;
    const leftMax = Array(n).fill(0);
    const rightMax = Array(n).fill(0);

    leftMax[0] = height[0];
    for (let i = 1; i < n; i++) {
        leftMax[i] = Math.max(leftMax[i - 1], height[i]);
    }

    rightMax[n - 1] = height[n - 1];
    for (let i = n - 2; i >= 0; i--) {
        rightMax[i] = Math.max(rightMax[i + 1], height[i]);
    }

    let maxWater = 0;
    for (let i = 0; i < n; i++) {
        const width = i;
        const currentHeight = Math.min(leftMax[i], rightMax[i]);
        maxWater = Math.max(maxWater, width * currentHeight);
    }

    return maxWater;
}

// Test
console.log(maxAreaWithPrecomputation([1, 8, 6, 2, 5, 4, 8, 3, 7])); // Output: 49
console.log(maxAreaWithPrecomputation([4, 3, 2, 1, 4])); // Output: 16
```

### Explanation:

1. **Objective:** Precompute the maximum heights from both ends to optimize area calculation.
2. **Approach:**
  - o Use two arrays:
    - $\text{leftMax}[i]$ : The maximum height from the left up to index  $i$ .
    - $\text{rightMax}[i]$ : The maximum height from the right up to index  $i$ .
  - o Use these arrays to compute the area for each index efficiently.
3. **Steps:**
  - o Precompute the left and right maximum heights.
  - o Use these values to calculate the area at each index and track the maximum area.

4. **Why it Works:** Precomputing the maximum heights allows efficient area calculation in a single pass.
5. **Time Complexity:**  $O(n)$  — Three passes through the array.
6. **Space Complexity:**  $O(n)$  — Space for the leftMax and rightMax arrays.

## 2. String Manipulation & Pattern Matching

### Detecting Repeated Character Patterns (Sliding Window Approach)

**101. Problem:** Given a string s and an integer k, you can replace at most k characters in the string so that the resulting string contains only one distinct character. Return the length of the longest substring with the same character you can achieve after replacement.

**Example:**

Input: s = "ABAB", k = 2

Output: 4

Explanation: Replace the two 'A's with 'B's to get "BBBB" or the two 'B's with 'A's to get "AAAA".

Input: s = "AABABBA", k = 1

Output: 4

Explanation: Replace the one 'A' in the middle with 'B' to get "AABBBA".

**Solution:**

```
function characterReplacement(s, k) {  
    let left = 0;  
    let maxFreq = 0;  
    let charCount = { };  
    let maxLength = 0;  
  
    for (let right = 0; right < s.length; right++) {  
        const char = s[right];  
        charCount[char] = (charCount[char] || 0) + 1;  
  
        maxFreq = Math.max(maxFreq, charCount[char]);  
  
        // Current window length is (right - left + 1)  
        // If the number of changes needed is greater than k, shrink the window  
        while (right - left + 1 - maxFreq > k) {  
            charCount[s[left]]--;  
            left++;  
        }  
  
        maxLength = Math.max(maxLength, right - left + 1);  
    }  
  
    return maxLength;  
}
```

```
// Test
console.log(characterReplacement("ABAB", 2)); // Output: 4
console.log(characterReplacement("AABABBA", 1)); // Output: 4
```

### **Explanation:**

1. **Objective:** Find the longest substring where you can replace at most k characters to make all characters in the substring the same.
2. **Approach:** Use the sliding window technique to keep track of the current window of characters and their frequencies.
3. **Steps:**
  - o Use a charCount object to count the frequency of characters in the current window.
  - o Keep track of the maximum frequency of any character in the window (maxFreq).
  - o If the number of characters in the window that need to be replaced (right - left + 1 - maxFreq) exceeds k, shrink the window from the left.
  - o Update the maximum length of the valid window.
4. **Why it Works:** The sliding window ensures we only traverse the string once, and adjusting the window guarantees that replacements stay within the allowed k.
5. **Time Complexity:** O(n) — Single traversal of the string.
6. **Space Complexity:** O(1) — Space for the character frequency map (limited to 26 letters).

### **Detecting Repeated Character Patterns (Optimized Frequency Update)**

**102. Problem: Find the length of the longest substring that can be formed by replacing at most k characters with the same character, while optimizing the way frequencies are updated.**

### **Example:**

Input: s = "AABABBA", k = 2  
 Output: 5  
 Explanation: Replace the two 'B's in the middle with 'A's to get "AAAAA".

### **Solution:**

```
function characterReplacementOptimized(s, k) {
  let left = 0;
  let maxFreq = 0;
  const charCount = Array(26).fill(0); // Store frequencies of characters (A-Z)

  const charToIndex = (char) => char.charCodeAt(0) - "A".charCodeAt(0);

  let maxLength = 0;

  for (let right = 0; right < s.length; right++) {
    const rightIndex = charToIndex(s[right]);
    charCount[rightIndex]++;
    if (charCount[rightIndex] > maxFreq) {
      maxFreq = charCount[rightIndex];
    }
    if (right - left + 1 - maxFreq > k) {
      const leftIndex = charToIndex(s[left]);
      charCount[leftIndex]--;
      left++;
    }
    maxLength = Math.max(maxLength, right - left + 1);
  }
  return maxLength;
}
```

```

maxFreq = Math.max(maxFreq, charCount[rightIndex]);

// Check if the window is valid
if (right - left + 1 - maxFreq > k) {
    const leftIndex = charToIndex(s[left]);
    charCount[leftIndex]--;
    left++;
}

maxLength = Math.max(maxLength, right - left + 1);
}

return maxLength;
}

// Test
console.log(characterReplacementOptimized("AABABBA", 2)); // Output: 5
console.log(characterReplacementOptimized("AAAB", 0)); // Output: 3

```

### Explanation:

1. **Objective:** Achieve the same result as before but with optimized frequency updates using an array for character counts.
2. **Approach:**
  - o Use an array of size 26 (charCount) to store character frequencies instead of an object.
  - o Map each character to an index in the array using its ASCII value.
3. **Steps:**
  - o Update the frequency of the current character in charCount.
  - o Track the maximum frequency (maxFreq) of any character in the window.
  - o If the replacements needed exceed k, shrink the window from the left.
  - o Update the maximum valid window size.
4. **Why it Works:** Using an array for frequencies improves performance by avoiding hash map operations.
5. **Time Complexity:** O(n) — Single traversal of the string.
6. **Space Complexity:** O(1) — Fixed-size array of 26 elements for character frequencies.

### Detecting Repeated Character Patterns (Exact Replacement Count)

**103. Problem:** Find the maximum length of a substring that can be achieved with exactly k character replacements.

#### Example:

Input: s = "AABABBA", k = 2  
Output: 5  
Explanation: Replace two 'B's with 'A's to get "AAAAA".

## Solution:

```
function characterReplacementExact(s, k) {
    let left = 0;
    let maxLength = 0;
    const charCount = Array(26).fill(0);
    const charToIndex = (char) => char.charCodeAt(0) - "A".charCodeAt(0);

    for (let right = 0; right < s.length; right++) {
        const rightIndex = charToIndex(s[right]);
        charCount[rightIndex]++;

        // Keep track of the number of characters to replace
        const replacementsNeeded = right - left + 1 - Math.max(...charCount);

        if (replacementsNeeded === k) {
            maxLength = Math.max(maxLength, right - left + 1);
        } else if (replacementsNeeded > k) {
            const leftIndex = charToIndex(s[left]);
            charCount[leftIndex]--;
            left++;
        }
    }

    return maxLength;
}

// Test
console.log(characterReplacementExact("AABABBA", 2)); // Output: 5
console.log(characterReplacementExact("ABAB", 1)); // Output: 3
```

## Explanation:

1. **Objective:** Calculate the maximum substring length with exactly  $k$  replacements.
2. **Approach:**
  - o Similar to the sliding window approach but explicitly checks for  $k$  replacements.
  - o Track the number of replacements needed to form the substring with the most frequent character.
3. **Steps:**
  - o Calculate the required replacements for the current window.
  - o If replacements equal  $k$ , update the maximum length.
  - o If replacements exceed  $k$ , shrink the window from the left.
4. **Why it Works:** By explicitly checking for exact  $k$  replacements, the solution guarantees correctness for specific constraints.
5. **Time Complexity:**  $O(n)$  — Single traversal of the string.

6. **Space Complexity:** O(1) — Fixed-size array of 26 elements for character counts.

### Detecting Repeated Character Patterns (Handle Case Insensitivity)

**104. Problem:** Extend the solution to handle both uppercase and lowercase letters in the input string.

**Example:**

Input: s = "AaBbAa", k = 2

Output: 6

Explanation: Replace two 'b' or 'B' with 'a' or 'A' to get "AAAAAA".

**Solution:**

```
function characterReplacementCaseInsensitive(s, k) {  
    let left = 0;  
    let maxFreq = 0;  
    const charCount = Array(26).fill(0);  
    const charToIndex = (char) =>  
        char.toUpperCase().charCodeAt(0) - "A".charCodeAt(0);  
  
    let maxLength = 0;  
  
    for (let right = 0; right < s.length; right++) {  
        const rightIndex = charToIndex(s[right]);  
        charCount[rightIndex]++;  
        maxFreq = Math.max(maxFreq, charCount[rightIndex]);  
  
        if (right - left + 1 - maxFreq > k) {  
            const leftIndex = charToIndex(s[left]);  
            charCount[leftIndex]--;  
            left++;  
        }  
  
        maxLength = Math.max(maxLength, right - left + 1);  
    }  
  
    return maxLength;  
}  
  
// Test  
console.log(characterReplacementCaseInsensitive("AaBbAa", 2)); // Output: 6  
console.log(characterReplacementCaseInsensitive("ABab", 1)); // Output: 4
```

**Explanation:**

1. **Objective:** Handle both uppercase and lowercase letters while finding the longest substring with up to k replacements.
2. **Approach:**
  - o Convert all characters to uppercase during frequency calculations.
  - o Use the same sliding window logic as in the previous solutions.
3. **Steps:**
  - o Normalize characters to uppercase before updating frequency counts.
  - o Proceed with the standard sliding window algorithm.
4. **Why it Works:** Case normalization ensures that the same character in different cases is treated as one, while sliding window efficiently tracks valid substrings.
5. **Time Complexity:** O(n) — Single traversal of the string.
6. **Space Complexity:** O(1) — Fixed-size array of 26 elements for character counts.

### Detecting Repeated Character Patterns (With Distinct Replacement Limits per Character)

**105. Problem:** Extend the solution to allow different replacement limits (k) for each character. For example, if k is provided as an object { A: 2, B: 1 }, then you can replace at most 2 occurrences of A and 1 occurrence of B in the string.

**Example:**

Input: s = "AABABBA", k = { A: 2, B: 1 }

Output: 5

Explanation: Replace one 'B' with 'A' and two 'A's with 'B' to get "AAAAA".

**Solution:**

```
function characterReplacementWithLimits(s, k) {
  let left = 0;
  let maxFreq = 0;
  const charCount = {};
  let maxLength = 0;

  for (let right = 0; right < s.length; right++) {
    const char = s[right];
    charCount[char] = (charCount[char] || 0) + 1;

    maxFreq = Math.max(maxFreq, charCount[char]);

    // Check the replacement limit for each character
    const replacementNeeded = right - left + 1 - maxFreq;
    const replacementLimit = Object.keys(charCount).reduce(
      (limit, key) =>
        limit +
        (charCount[key] > (k[key] || Infinity)
          ? charCount[key] - (k[key] || Infinity)
          : 0),
    );
  }
}
```

```

        0
    );

    if (replacementNeeded > replacementLimit) {
        charCount[s[left]]--;
        if (charCount[s[left]] === 0) delete charCount[s[left]];
        left++;
    }

    maxLength = Math.max(maxLength, right - left + 1);
}

return maxLength;
}

// Test
console.log(characterReplacementWithLimits("AABABBA", { A: 2, B: 1 })); // Output: 5
console.log(characterReplacementWithLimits("ABAB", { A: 1, B: 2 })); // Output: 4

```

### **Explanation:**

1. **Objective:** Allow different replacement limits per character and find the longest substring with valid replacements.
2. **Approach:**
  - o Use a charCount object to track character frequencies in the current window.
  - o Calculate the maximum allowable replacements for each character using the provided k limits.
  - o Shrink the window when the number of replacements exceeds the allowable limits.
3. **Steps:**
  - o Traverse the string, updating the character count for the current window.
  - o Compute the replacement needs and compare them with the replacement limits.
  - o Adjust the left pointer to shrink the window when the replacement limits are violated.
4. **Why it Works:** This approach dynamically calculates valid windows based on character-specific replacement constraints.
5. **Time Complexity:**  $O(n)$  — Single traversal of the string.
6. **Space Complexity:**  $O(c)$  — Space for the charCount object, where c is the number of distinct characters in s.

### **Detecting Repeated Character Patterns (Allow Partial Replacement)**

**106. Problem: Modify the solution to allow partially replacing characters to meet the limit. For instance, if only 1 replacement is possible out of 3 needed, replace only 1 character and calculate the valid substring length.**

### **Example:**

Input: s = "AABABBA", k = 1

Output: 4

Explanation: Replace the one 'A' in the middle with 'B' to get "AABBBA".

**Solution:**

```
function characterReplacementPartial(s, k) {  
    let left = 0;  
    let maxFreq = 0;  
    const charCount = {};  
    let maxLength = 0;  
  
    for (let right = 0; right < s.length; right++) {  
        const char = s[right];  
        charCount[char] = (charCount[char] || 0) + 1;  
  
        maxFreq = Math.max(maxFreq, charCount[char]);  
  
        // Current window length  
        const windowLength = right - left + 1;  
  
        // If replacements exceed the limit, shrink the window  
        if (windowLength - maxFreq > k) {  
            charCount[s[left]]--;  
            left++;  
        }  
  
        maxLength = Math.max(maxLength, right - left + 1);  
    }  
  
    return maxLength;  
}  
  
// Test  
console.log(characterReplacementPartial("AABABBA", 1)); // Output: 4  
console.log(characterReplacementPartial("ABAB", 2)); // Output: 4
```

**Explanation:**

1. **Objective:** Allow partial replacements to meet the constraints and calculate the maximum substring length.
2. **Approach:**
  - o Use the sliding window technique to keep track of the current valid substring.
  - o Shrink the window when the number of replacements exceeds the allowable limit.
3. **Steps:**
  - o Maintain a character frequency map and track the maximum frequency character in the window.

- Calculate the number of replacements needed for the current window and compare it with k.
  - Adjust the left pointer when replacements exceed the limit.
4. **Why it Works:** By shrinking the window dynamically, the solution ensures the maximum valid substring is found.
5. **Time Complexity:**  $O(n)$  — Single traversal of the string.
6. **Space Complexity:**  $O(c)$  — Space for the charCount object, where c is the number of distinct characters in s.

### Detecting Repeated Character Patterns (Handle Unicode Characters)

**107. Problem:** Extend the solution to handle strings containing Unicode characters, such as emojis or non-English alphabets.

**Example:**

Input: s = " AppleWebKit ", k = 1  
 Output: 5  
 Explanation: Replace one with to get " ".

**Solution:**

```
function characterReplacementUnicode(s, k) {
  let left = 0;
  let maxFreq = 0;
  const charCount = {};
  let maxLength = 0;

  for (let right = 0; right < s.length; right++) {
    const char = s[right];
    charCount[char] = (charCount[char] || 0) + 1;

    maxFreq = Math.max(maxFreq, charCount[char]);

    if (right - left + 1 - maxFreq > k) {
      charCount[s[left]]--;
      left++;
    }
  }

  maxLength = Math.max(maxLength, right - left + 1);
}

return maxLength;
}

// Test
console.log(characterReplacementUnicode(" AppleWebKit ", 1)); // Output: 5
```

```
console.log(characterReplacementUnicode("ພພບພບ", 2)); // Output: 6
```

### Explanation:

1. **Objective:** Handle Unicode characters while finding the longest substring with up to k replacements.
2. **Approach:**
  - o Use a character frequency map to store counts for all Unicode characters.
  - o Apply the sliding window technique to dynamically calculate valid substrings.
3. **Steps:**
  - o Update the frequency map for the current character.
  - o Calculate the maximum frequency and adjust the window if replacements exceed k.
4. **Why it Works:** Unicode characters are treated like any other characters, and the solution dynamically adjusts to handle their presence.
5. **Time Complexity:** O(n) — Single traversal of the string.
6. **Space Complexity:** O(c) — Space for the charCount object, where c is the number of distinct characters in s.

### Detecting Repeated Character Patterns (Find All Valid Substrings)

**108. Problem: Return all substrings of the input string s where the length of the substring is the maximum that can be achieved with up to k replacements.**

#### Example:

Input: s = "AABABBA", k = 1

Output: ["AABA", "ABAB", "BABB"]

Explanation: These substrings have the maximum length of 4 and can be achieved with 1 replacement.

#### Solution:

```
function findAllValidSubstrings(s, k) {  
    let left = 0;  
    let maxFreq = 0;  
    const charCount = {};  
    let maxLength = 0;  
    const validSubstrings = [];  
  
    for (let right = 0; right < s.length; right++) {  
        const char = s[right];  
        charCount[char] = (charCount[char] || 0) + 1;  
  
        maxFreq = Math.max(maxFreq, charCount[char]);  
        if (maxFreq === k) {  
            validSubstrings.push(s.substring(left, right + 1));  
        }  
    }  
    return validSubstrings;  
}
```

```

while (right - left + 1 - maxFreq > k) {
    charCount[s[left]]--;
    left++;
}

const currentLength = right - left + 1;
if (currentLength > maxLength) {
    maxLength = currentLength;
    validSubstrings.length = 0; // Clear previous substrings
    validSubstrings.push(s.slice(left, right + 1));
} else if (currentLength === maxLength) {
    validSubstrings.push(s.slice(left, right + 1));
}
}

return validSubstrings;
}

// Test
console.log(findAllValidSubstrings("AABABBA", 1)); // Output: ["AABA", "ABAB",
"BABB"]
console.log(findAllValidSubstrings("ABAB", 2)); // Output: ["ABAB"]

```

### **Explanation:**

1. **Objective:** Identify all substrings that meet the constraints of  $k$  replacements and have the maximum possible length.
2. **Approach:**
  - o Use the sliding window technique to dynamically track valid substrings.
  - o Track the maximum length and store substrings of that length.
3. **Steps:**
  - o Update character frequencies and calculate the maximum frequency in the window.
  - o Adjust the left pointer to shrink the window if replacements exceed  $k$ .
  - o Update the list of valid substrings whenever a new maximum length is found.
4. **Why it Works:** By dynamically updating the valid substrings list, we ensure all maximum-length substrings are included.
5. **Time Complexity:**  $O(n)$  — Single traversal of the string.
6. **Space Complexity:**  $O(c + m)$  — Space for the `charCount` object and the valid substrings list ( $m$  substrings).

### **Detecting Repeated Character Patterns (Count the Number of Valid Substrings)**

**109. Problem:** Count how many substrings of the input string  $s$  can achieve the maximum possible length with up to  $k$  replacements.

### **Example:**

Input: s = "AABABBA", k = 1

Output: 3

Explanation: There are 3 substrings ("AABA", "ABAB", "BABB") with the maximum length of 4.

### Solution:

```
function countValidSubstrings(s, k) {  
    let left = 0;  
    let maxFreq = 0;  
    const charCount = {};  
    let maxLength = 0;  
    let count = 0;  
  
    for (let right = 0; right < s.length; right++) {  
        const char = s[right];  
        charCount[char] = (charCount[char] || 0) + 1;  
  
        maxFreq = Math.max(maxFreq, charCount[char]);  
  
        while (right - left + 1 - maxFreq > k) {  
            charCount[s[left]]--;  
            left++;  
        }  
  
        const currentLength = right - left + 1;  
        if (currentLength > maxLength) {  
            maxLength = currentLength;  
            count = 1; // Reset the count for new maximum length  
        } else if (currentLength === maxLength) {  
            count++;  
        }  
    }  
  
    return count;  
}  
  
// Test  
console.log(countValidSubstrings("AABABBA", 1)); // Output: 3  
console.log(countValidSubstrings("ABAB", 2)); // Output: 1
```

### Explanation:

1. **Objective:** Count the number of valid substrings that can achieve the maximum possible length with up to k replacements.

2. **Approach:**

- Use the sliding window technique to track the length of the valid substring dynamically.
  - Update the count whenever a substring of maximum length is encountered.
3. **Steps:**
- Traverse the string and calculate the maximum frequency in the window.
  - Adjust the left pointer if replacements exceed k.
  - Track the count of substrings of maximum length.
4. **Why it Works:** By counting substrings dynamically, we efficiently determine the number of valid substrings without redundant computations.
5. **Time Complexity:**  $O(n)$  — Single traversal of the string.
6. **Space Complexity:**  $O(c)$  — Space for the charCount object, where c is the number of distinct characters.

### **Detecting Repeated Character Patterns (With Custom Replacement Costs)**

**110. Problem: If replacing each character has a different cost, find the maximum length of a substring where the total cost of replacements does not exceed k.**

**Example:**

Input: s = "AABABBA", k = 5, costs = { A: 2, B: 3 }

Output: 5

Explanation: Replace two 'B's with 'A's, costing  $3 + 3 = 6$ , which is within the allowed `k = 5`.

**Solution:**

```
function characterReplacementWithCosts(s, k, costs) {
  let left = 0;
  let totalCost = 0;
  const charCount = {};
  let maxLength = 0;

  for (let right = 0; right < s.length; right++) {
    const char = s[right];
    charCount[char] = (charCount[char] || 0) + 1;

    // Calculate the replacement cost
    totalCost += costs[char] || 0;

    while (totalCost > k) {
      totalCost -= costs[s[left]] || 0;
      charCount[s[left]]--;
      left++;
    }

    maxLength = Math.max(maxLength, right - left + 1);
  }
}
```

```

    return maxLength;
}

// Test
console.log(characterReplacementWithCosts("AABABBA", 5, { A: 2, B: 3 })); // Output: 5
console.log(characterReplacementWithCosts("ABAB", 4, { A: 1, B: 2 })); // Output: 4

```

### **Explanation:**

1. **Objective:** Allow custom replacement costs for each character while finding the longest valid substring.
2. **Approach:**
  - o Use a sliding window technique while tracking the total replacement cost for the current window.
  - o Adjust the left pointer to ensure the total cost does not exceed k.
3. **Steps:**
  - o Update the total replacement cost dynamically for each character in the window.
  - o Shrink the window if the total cost exceeds k.
  - o Track the maximum valid window length.
4. **Why it Works:** By dynamically tracking the replacement costs, the solution ensures the constraints are satisfied efficiently.
5. **Time Complexity:** O(n) — Single traversal of the string.
6. **Space Complexity:** O(c) — Space for the charCount object and the costs map.

### **Identifying Anagrams in Word Sets (Basic Check)**

**111. Problem: Given two strings s and t, return true if t is an anagram of s, and false otherwise.**

**An anagram is a word or phrase formed by rearranging the letters of another, using all the original letters exactly once.**

### **Example:**

Input: (s = "anagram"), (t = "nagaram");  
Output: **true**;

Input: (s = "rat"), (t = "car");  
Output: **false**;

### **Solution:**

```

function isAnagram(s, t) {
  if (s.length !== t.length) return false;

```

```

const count = { };

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

for (const char of t) {
    if (!count[char]) return false;
    count[char]--;
}

return true;
}

// Test
console.log(isAnagram("anagram", "nagaram")); // Output: true
console.log(isAnagram("rat", "car")); // Output: false

```

### **Explanation:**

1. **Objective:** Determine if one string is an anagram of the other.
2. **Approach:**
  - o Count the frequency of each character in the first string using a hash map.
  - o Iterate through the second string and decrease the count for each character.
  - o If a character is missing or overused, return false.
3. **Steps:**
  - o If the lengths of the strings differ, return false immediately.
  - o Build a frequency map for the first string.
  - o Check the second string against this map.
4. **Why it Works:** Comparing character frequencies ensures all characters are matched correctly.
5. **Time Complexity:**  $O(n)$  — Single traversal of both strings.
6. **Space Complexity:**  $O(c)$  — Space for the frequency map, where  $c$  is the number of unique characters.

### **Identifying Anagrams in Word Sets (Sort-Based)**

#### **112. Problem: Use sorting to determine if two strings are anagrams.**

##### **Example:**

Input: ( $s = \text{"listen"}$ ), ( $t = \text{"silent"}$ );  
Output: **true**;

Input: ( $s = \text{"hello"}$ ), ( $t = \text{"world"}$ );  
Output: **false**;

### Solution:

```
function isAnagramSorted(s, t) {  
    if (s.length !== t.length) return false;  
  
    return s.split("").sort().join("") === t.split("").sort().join("");  
}  
  
// Test  
console.log(isAnagramSorted("listen", "silent")); // Output: true  
console.log(isAnagramSorted("hello", "world")); // Output: false
```

### Explanation:

1. **Objective:** Check if two strings are anagrams by comparing their sorted versions.
2. **Approach:**
  - o Sort the characters of both strings.
  - o Compare the sorted strings for equality.
3. **Steps:**
  - o Split the strings into arrays, sort them, and rejoin them.
  - o If the sorted versions match, the strings are anagrams.
4. **Why it Works:** Sorting rearranges characters in a consistent order, making it easy to compare equality.
5. **Time Complexity:**  $O(n \log n)$  — Due to the sorting operation.
6. **Space Complexity:**  $O(n)$  — Space for the sorted arrays.

### Identifying Anagrams in Word Sets (Case-Insensitive)

#### 113. Problem: Modify the solution to handle case-insensitive comparisons.

### Example:

```
Input: (s = "Listen"), (t = "Silent");  
Output: true;
```

```
Input: (s = "Hello"), (t = "World");  
Output: false;
```

### Solution:

```
function isAnagramCaseInsensitive(s, t) {  
    if (s.length !== t.length) return false;  
  
    const normalize = (str) => str.toLowerCase().split("").sort().join("");  
    return normalize(s) === normalize(t);  
}
```

```
// Test
console.log(isAnagramCaseInsensitive("Listen", "Silent")); // Output: true
console.log(isAnagramCaseInsensitive("Hello", "World")); // Output: false
```

### Explanation:

1. **Objective:** Handle case-insensitive comparisons for anagram checking.
2. **Approach:**
  - o Convert both strings to lowercase before sorting and comparing.
3. **Steps:**
  - o Normalize the strings by converting to lowercase and sorting.
  - o Compare the normalized strings for equality.
4. **Why it Works:** Lowercasing ensures case insensitivity, while sorting ensures consistent ordering.
5. **Time Complexity:**  $O(n \log n)$  — Due to sorting.
6. **Space Complexity:**  $O(n)$  — Space for the normalized strings.

## Identifying Anagrams in Word Sets (With Unicode Characters)

### 114. Problem: Check if two strings with Unicode characters are anagrams.

#### Example:

```
Input: (s = "déjà vu"), (t = "vu déjà");
Output: true;
```

```
Input: (s = "déjà");
Output: false;
```

#### Solution:

```
function isAnagramUnicode(s, t) {
  const normalize = (str) =>
    str.replace(/\s+/g, "").toLowerCase().split("").sort().join("");
  return normalize(s) === normalize(t);
}

// Test
console.log(isAnagramUnicode("déjà vu", "vu déjà")); // Output: true
console.log(isAnagramUnicode("déjà", "java")); // Output: false
```

### Explanation:

1. **Objective:** Handle Unicode characters and ignore spaces when checking for anagrams.
2. **Approach:**
  - o Normalize the strings by:

- Removing spaces.
  - Converting to lowercase.
  - Sorting characters.
  - Compare the normalized strings.
3. **Steps:**
- Use a regular expression to remove spaces.
  - Apply lowercase conversion and sorting.
  - Compare the normalized versions.
4. **Why it Works:** This ensures consistent processing of Unicode characters and spaces.
5. **Time Complexity:**  $O(n \log n)$  — Due to sorting.
6. **Space Complexity:**  $O(n)$  — Space for the normalized strings.

### **Identifying Anagrams in Word Sets (Character Frequency Array for English Letters)**

**115. Problem: Check if two strings are anagrams by using a fixed-size frequency array for English letters (a-z).**

#### **Example:**

Input: (*s* = "anagram"), (*t* = "nagaram");  
 Output: **true**;

Input: (*s* = "rat"), (*t* = "car");  
 Output: **false**;

#### **Solution:**

```
function isAnagramUsingArray(s, t) {
  if (s.length !== t.length) return false;

  const charCount = Array(26).fill(0);

  for (const char of s) {
    charCount[char.charCodeAt(0) - "a".charCodeAt(0)]++;
  }

  for (const char of t) {
    const index = char.charCodeAt(0) - "a".charCodeAt(0);
    if (charCount[index] === 0) return false;
    charCount[index]--;
  }

  return true;
}

// Test
console.log(isAnagramUsingArray("anagram", "nagaram")); // Output: true
console.log(isAnagramUsingArray("rat", "car")); // Output: false
```

### **Explanation:**

1. **Objective:** Use a fixed-size array to store character frequencies for the English alphabet.
2. **Approach:**
  - o Map each character to an index based on its ASCII value ( $a = 0, b = 1, \dots, z = 25$ ).
  - o Increment counts for  $s$  and decrement counts for  $t$ .
  - o If any count goes below zero,  $t$  is not an anagram of  $s$ .
3. **Steps:**
  - o Iterate through  $s$  to populate the frequency array.
  - o Iterate through  $t$  and adjust the frequency array.
  - o If all counts are zero after processing both strings, they are anagrams.
4. **Why it Works:** A fixed-size array provides efficient and constant-time access for frequency adjustments.
5. **Time Complexity:**  $O(n)$  — Single traversal of both strings.
6. **Space Complexity:**  $O(1)$  — Fixed-size array of 26 elements.

### **Identifying Anagrams in Word Sets (Hash Map for Arbitrary Characters)**

**116. Problem: Handle strings containing arbitrary characters (not limited to the English alphabet).**

#### **Example:**

Input: ( $s = "déjà vu"$ ), ( $t = "vu déjà"$ );  
Output: **true**;

Input: ( $s = "hello"$ ), ( $t = "world"$ );  
Output: **false**;

#### **Solution:**

```
function isAnagramUsingHashMap(s, t) {  
    if (s.length !== t.length) return false;  
  
    const charCount = {};  
  
    for (const char of s) {  
        charCount[char] = (charCount[char] || 0) + 1;  
    }  
  
    for (const char of t) {  
        if (!charCount[char]) return false;  
        charCount[char]--;  
    }  
  
    return Object.values(charCount).every((count) => count === 0);  
}
```

```
// Test
console.log(isAnagramUsingHashMap("déjà vu", "vu déjà")); // Output: true
console.log(isAnagramUsingHashMap("hello", "world")); // Output: false
```

### Explanation:

1. **Objective:** Check anagrams for strings containing arbitrary characters.
2. **Approach:**
  - o Use a hash map to store character frequencies for s.
  - o Decrement counts for characters in t.
  - o Ensure all counts are zero after processing both strings.
3. **Steps:**
  - o Populate the frequency map for s.
  - o Adjust the frequency map for t and check for negative counts.
  - o Ensure all values in the map are zero.
4. **Why it Works:** The hash map dynamically handles any characters, including Unicode and special characters.
5. **Time Complexity:** O(n) — Single traversal of both strings.
6. **Space Complexity:** O(c) — Space for the hash map, where c is the number of unique characters.

### Identifying Anagrams in Word Sets (Ignoring Spaces and Case)

#### 117. Problem: Check if two strings are anagrams while ignoring spaces and case.

##### Example:

```
Input: (s = "A gentleman"), (t = "Elegant man");
Output: true;
```

```
Input: (s = "Hello"), (t = "Oleh");
Output: false;
```

##### Solution:

```
function isAnagramIgnoreSpacesAndCase(s, t) {
  const normalize = (str) =>
    str.replace(/\s+/g, "").toLowerCase().split("").sort().join("");
  
  return normalize(s) === normalize(t);
}

// Test
console.log(isAnagramIgnoreSpacesAndCase("A gentleman", "Elegant man")); // Output:
true
console.log(isAnagramIgnoreSpacesAndCase("Hello", "Oleh")); // Output: false
```

## Explanation:

1. **Objective:** Ensure the comparison ignores spaces and character case.
2. **Approach:**
  - o Normalize strings by removing spaces, converting to lowercase, and sorting characters.
  - o Compare the normalized versions for equality.
3. **Steps:**
  - o Use replace with a regular expression to remove spaces.
  - o Convert to lowercase and sort the characters.
  - o Compare the sorted strings.
4. **Why it Works:** Normalizing strings ensures consistent processing regardless of spaces or case differences.
5. **Time Complexity:**  $O(n \log n)$  — Due to sorting.
6. **Space Complexity:**  $O(n)$  — Space for the normalized strings.

## Identifying Anagrams in Word Sets (Count Unicode Characters)

**118. Problem: Check if two strings are anagrams for Unicode characters, including emojis.**

### Example:

Input: (s = "𠮷 𠮷 𠮷"), (t = "𠮷 𠮷 𠮷");  
Output: true;

Input: (s = "𠮷 𠮷 ߂"), (t = "߂ ߂ 𠮷");  
Output: false;

### Solution:

```
function isAnagramUnicodeCount(s, t) {  
    if (s.length !== t.length) return false;  
  
    const charCount = {};  
  
    for (const char of s) {  
        charCount[char] = (charCount[char] || 0) + 1;  
    }  
  
    for (const char of t) {  
        if (!charCount[char]) return false;  
        charCount[char]--;  
    }  
  
    return Object.values(charCount).every((count) => count === 0);
```

```

}

// Test
console.log(isAnagramUnicodeCount("𠮷𠮷𠮷", "𠮷𠮷𠮷")); // Output: true
console.log(isAnagramUnicodeCount("𠮷𠮷𠮷", "𠮷𠮷𠮷")); // Output: false

```

### Explanation:

1. **Objective:** Handle Unicode characters when checking for anagrams.
2. **Approach:**
  - o Use a hash map to count character frequencies for Unicode strings.
  - o Ensure counts are balanced after processing both strings.
3. **Steps:**
  - o Populate the hash map with frequencies from s.
  - o Adjust the map with frequencies from t.
  - o Check if all counts are zero.
4. **Why it Works:** The hash map dynamically supports all Unicode characters.
5. **Time Complexity:**  $O(n)$  — Single traversal of both strings.
6. **Space Complexity:**  $O(c)$  — Space for the hash map, where c is the number of unique characters.

### Identifying Anagrams in Word Sets (With Prime Multiplication Method)

**119. Problem: Check if two strings are anagrams by representing each character as a unique prime number and using their product as a hash.**

#### Example:

```

Input: (s = "anagram"), (t = "nagaram");
Output: true;

Input: (s = "rat"), (t = "car");
Output: false;

```

#### Solution:

```

function isAnagramPrimeHash(s, t) {
  if (s.length !== t.length) return false;

  // Assign prime numbers to each character (a-z)
  const primes = [
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
    73, 79, 83, 89, 97, 101,
  ];

  const charToPrime = (char) => primes[char.charCodeAt(0) - "a".charCodeAt(0)];

```

```

const hash = (str) =>
  str.split("").reduce((product, char) => product * charToPrime(char), 1);

  return hash(s) === hash(t);
}

// Test
console.log(isAnagramPrimeHash("anagram", "nagaram")); // Output: true
console.log(isAnagramPrimeHash("rat", "car")); // Output: false

```

### Explanation:

1. **Objective:** Use the unique property of prime numbers for character mapping to determine anagrams.
2. **Approach:**
  - o Assign each character a unique prime number.
  - o Calculate the product of the prime numbers for the characters in both strings.
  - o If the products match, the strings are anagrams.
3. **Steps:**
  - o Create an array of prime numbers for characters a-z.
  - o Compute the hash for both strings as the product of primes.
  - o Compare the two hashes.
4. **Why it Works:** The product of primes uniquely represents each set of characters, as prime factorization is unique.
5. **Time Complexity:** O(n) — Single traversal of both strings.
6. **Space Complexity:** O(1) — Fixed space for the primes array.

### Identifying Anagrams in Word Sets (Count Uppercase and Lowercase Separately)

**120. Problem: Check if two strings are anagrams, considering uppercase and lowercase characters as distinct.**

### Example:

Input: (s = "aA"), (t = "Aa");  
 Output: true;

Input: (s = "aA"), (t = "aa");  
 Output: false;

### Solution:

```

function isAnagramCaseSensitive(s, t) {
  if (s.length !== t.length) return false;

  const charCount = {};
  for (const char of s) {
    if (char in charCount) {
      charCount[char] += 1;
    } else {
      charCount[char] = 1;
    }
  }

  for (const char of t) {
    if (!charCount[char]) {
      return false;
    } else {
      charCount[char] -= 1;
    }
  }

  return Object.values(charCount).every(count => count === 0);
}

```

```

        charCount[char] = (charCount[char] || 0) + 1;
    }

    for (const char of t) {
        if (!charCount[char]) return false;
        charCount[char]--;
    }

    return Object.values(charCount).every((count) => count === 0);
}

// Test
console.log(isAnagramCaseSensitive("aA", "Aa")); // Output: true
console.log(isAnagramCaseSensitive("aA", "aa")); // Output: false

```

### Explanation:

1. **Objective:** Treat uppercase and lowercase characters as distinct when checking for anagrams.
2. **Approach:**
  - o Use a hash map to count character frequencies, differentiating between uppercase and lowercase.
  - o Ensure all counts match between the two strings.
3. **Steps:**
  - o Populate the hash map with frequencies from s.
  - o Adjust the map with frequencies from t.
  - o Check if all counts are zero.
4. **Why it Works:** Separate treatment of cases ensures correct anagram validation for case-sensitive strings.
5. **Time Complexity:** O(n) — Single traversal of both strings.
6. **Space Complexity:** O(c) — Space for the hash map, where c is the number of unique characters.

## Group Anagrams (Basic Approach Using Sorting)

### 121. Problem: Group all anagrams from the given array of strings strs into subarrays.

#### Example:

Input: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]  
Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]  
Explanation: Words with the same sorted characters are grouped together.

#### Solution:

```

function groupAnagrams(strs) {
    const map = new Map();

```

```

for (const str of strs) {
  const sorted = str.split("").sort().join("");
  if (!map.has(sorted)) {
    map.set(sorted, []);
  }
  map.get(sorted).push(str);
}

return Array.from(map.values());
}

// Test
console.log(groupAnagrams(["eat", "tea", "tan", "ate", "nat", "bat"]));
// Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]
console.log(groupAnagrams([""]));
// Output: [[]]
console.log(groupAnagrams(["a"]));
// Output: [["a"]]

```

### Explanation:

1. **Objective:** Group all strings with the same characters together.
2. **Approach:**
  - o Sort each string to create a unique key for its anagram group.
  - o Store the original strings in a hash map using the sorted string as the key.
  - o Return all values from the hash map.
3. **Steps:**
  - o Loop through the strings and sort each one.
  - o Add each string to the map under its sorted key.
  - o Collect all groups from the map.
4. **Why it Works:** Sorting rearranges characters in a consistent order, so anagrams will share the same sorted string.
5. **Time Complexity:**  $O(n * k \log k)$ , where  $n$  is the number of strings, and  $k$  is the average length of a string (due to sorting).
6. **Space Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

### Group Anagrams (Using Character Frequency)

**122. Problem: Group all anagrams from the array without sorting, by using a frequency count of characters.**

Input: strs = ["eat", "tea", "tan", "ate", "nat", "bat"];  
Output: [[ "eat", "tea", "ate"], [ "tan", "nat"], [ "bat"]];

### Solution:

```

function groupAnagramsByFrequency(strs) {
  const map = new Map();

  for (const str of strs) {
    const count = Array(26).fill(0);

    for (const char of str) {
      count[char.charCodeAt(0) - "a".charCodeAt(0)]++;
    }

    const key = count.join("#"); // Create a unique key for the frequency count
    if (!map.has(key)) {
      map.set(key, []);
    }
    map.get(key).push(str);
  }

  return Array.from(map.values());
}

// Test
console.log(
  groupAnagramsByFrequency(["eat", "tea", "tan", "ate", "nat", "bat"])
);
// Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]
console.log(groupAnagramsByFrequency([""]));
// Output: [[]]
console.log(groupAnagramsByFrequency(["a"]));
// Output: [["a"]]

```

### **Explanation:**

1. **Objective:** Avoid sorting by using a frequency count for each string.
2. **Approach:**
  - o Count the frequency of each character for each string.
  - o Use the frequency array as a key in the hash map.
3. **Steps:**
  - o Create an array of size 26 to represent character counts for each string.
  - o Convert the frequency array to a string key and store it in the map.
  - o Group strings with the same frequency array.
4. **Why it Works:** Anagrams have identical character frequencies, so they can be grouped using the frequency array.
5. **Time Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.
6. **Space Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

### **Group Anagrams (Case-Insensitive Approach)**

### 123. Problem: Group anagrams while ignoring the case of the characters in the strings.

**Example:**

```
Input: strs = ["Eat", "Tea", "tan", "Ate", "nat", "Bat"]
Output: [["Eat", "Tea", "Ate"], ["tan", "nat"], ["Bat"]]
Explanation: Anagrams are grouped together, case-insensitively.
```

**Solution:**

```
function groupAnagramsCaseInsensitive(strs) {
  const map = new Map();

  for (const str of strs) {
    const sorted = str.toLowerCase().split("").sort().join("");
    if (!map.has(sorted)) {
      map.set(sorted, []);
    }
    map.get(sorted).push(str);
  }

  return Array.from(map.values());
}

// Test
console.log(
  groupAnagramsCaseInsensitive(["Eat", "Tea", "tan", "Ate", "nat", "Bat"])
);
// Output: [["Eat", "Tea", "Ate"], ["tan", "nat"], ["Bat"]]
console.log(groupAnagramsCaseInsensitive([""]));
// Output: []
console.log(groupAnagramsCaseInsensitive(["a", "A"]));
// Output: [["a", "A"]]
```

**Explanation:**

1. **Objective:** Group strings into anagram groups, ignoring case differences.
2. **Approach:**
  - o Convert each string to lowercase before sorting.
  - o Use the sorted, lowercase string as the key in the hash map.
3. **Steps:**
  - o Normalize each string by converting it to lowercase and sorting its characters.
  - o Group strings with the same normalized key.
4. **Why it Works:** Lowercasing ensures consistency across different cases while sorting ensures identical characters are grouped together.
5. **Time Complexity:**  $O(n * k \log k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

6. **Space Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

### Group Anagrams (Using Unicode Characters)

**124. Problem:** Group anagrams from a list of strings containing Unicode characters, including emojis.

**Example:**

```
Input: strs = ["𠮷𠮷", "𠮷𠮷", "𠮷𠮷", "𠮷𠮷"]
Output: [["𠮷𠮷"], ["𠮷𠮷"], ["𠮷𠮷"]]
Explanation: Anagrams are grouped by their characters, including Unicode symbols.
```

**Solution:**

```
function groupAnagramsUnicode(strs) {
  const map = new Map();

  for (const str of strs) {
    const sorted = Array.from(str).sort().join("");
    if (!map.has(sorted)) {
      map.set(sorted, []);
    }
    map.get(sorted).push(str);
  }

  return Array.from(map.values());
}

// Test
console.log(groupAnagramsUnicode(["𠮷𠮷", "𠮷𠮷", "𠮷𠮷", "𠮷𠮷"]));
// Output: [["𠮷𠮷"], ["𠮷𠮷"], ["𠮷𠮷"]]
console.log(groupAnagramsUnicode([]));
// Output: []
console.log(groupAnagramsUnicode(["𠮷"]));
// Output: [[𠮷]]
```

**Explanation:**

1. **Objective:** Group anagrams containing Unicode characters.

2. **Approach:**

- o Convert each string into an array of its characters (including Unicode).
- o Sort the array to create a consistent key for anagrams.
- o Group strings with the same sorted key.

3. **Steps:**
  - o Use Array.from(str) to handle Unicode characters properly.
  - o Sort the array and join it back into a string.
  - o Use the sorted string as the key in a hash map.
4. **Why it Works:** Sorting ensures that characters (including Unicode) are consistently ordered for anagrams.
5. **Time Complexity:**  $O(n * k \log k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.
6. **Space Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

### **Group Anagrams (Handle Large Input Efficiently)**

**125. Problem: Optimize the solution to handle large input arrays with very long strings efficiently.**

**Example:**

```
Input: strs = ["a".repeat(100), "a".repeat(100), "b".repeat(100)];
Output: [[["a".repeat(100), "a".repeat(100)], ["b".repeat(100)]];
```

**Solution:**

```
function groupAnagramsLargeInput(strs) {
  const map = new Map();

  for (const str of strs) {
    const count = Array(26).fill(0);
    for (const char of str) {
      count[char.charCodeAt(0) - "a".charCodeAt(0)]++;
    }

    const key = count.join("#"); // Create a compact key for large strings
    if (!map.has(key)) {
      map.set(key, []);
    }
    map.get(key).push(str);
  }

  return Array.from(map.values());
}

// Test
console.log(
  groupAnagramsLargeInput(["a".repeat(100), "a".repeat(100), "b".repeat(100)])
);
// Output: [[["a".repeat(100), "a".repeat(100)], ["b".repeat(100)]]]
```

## Explanation:

1. **Objective:** Optimize for large strings by avoiding sorting.
2. **Approach:**
  - o Use a frequency count array to represent each string.
  - o Create a compact key from the frequency array using join('#').
  - o Use the frequency key to group anagrams in a hash map.
3. **Steps:**
  - o Count character frequencies for each string.
  - o Use the frequency array as a unique identifier for anagrams.
  - o Group strings with the same frequency key.
4. **Why it Works:** Counting characters avoids sorting large strings and provides a unique representation for anagrams.
5. **Time Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.
6. **Space Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

## Group Anagrams (Using Prime Product as Key)

**126. Problem:** **Group anagrams by assigning each character a unique prime number and using the product of these primes as a hash key.**

### Example:

```
Input: strs = ["eat", "tea", "tan", "ate", "nat", "bat"];
Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]];
```

### Solution:

```
function groupAnagramsPrimeProduct(strs) {
  const primeNumbers = [
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
    73, 79, 83, 89, 97, 101,
  ];

  const charToPrime = (char) =>
    primeNumbers[char.charCodeAt(0) - "a".charCodeAt(0)];
  const map = new Map();

  for (const str of strs) {
    let product = 1;
    for (const char of str) {
      product *= charToPrime(char);
    }
    if (!map.has(product)) {
      map.set(product, []);
    }
    map.get(product).push(str);
  }
}
```

```

    }

    return Array.from(map.values());
}

// Test
console.log(
  groupAnagramsPrimeProduct(["eat", "tea", "tan", "ate", "nat", "bat"])
);
// Output: [[{"eat": 101, "tea": 101, "tan": 101}, {"ate": 101, "nat": 101, "bat": 101}]]
console.log(groupAnagramsPrimeProduct([""]));
// Output: [[{"": 1}]]
console.log(groupAnagramsPrimeProduct(["a"]));
// Output: [{"a": 2}]]
```

### **Explanation:**

1. **Objective:** Use the unique properties of prime numbers to represent each string.
2. **Approach:**
  - o Assign each character a unique prime number (a = 2, b = 3, ..., z = 101).
  - o Compute the product of primes for the characters in each string.
  - o Use this product as a unique hash key to group anagrams.
3. **Steps:**
  - o Calculate the product of primes for each string.
  - o Use the product as a key in a hash map to group anagrams.
4. **Why it Works:** The product of primes is unique for every combination of characters, so anagrams will have the same product.
5. **Time Complexity:**  $O(n * k)$ , where n is the number of strings, and k is the average length of the strings.
6. **Space Complexity:**  $O(n * k)$ , where n is the number of strings, and k is the average length of the strings.

### **Group Anagrams (With Empty Strings Handling)**

#### **127. Problem: Handle cases where the input array contains multiple empty strings.**

##### **Example:**

Input: strs = ["", "eat", "tea", "", "bat", "tab"];  
Output: [  
 ['', ''],  
 ['eat', 'tea'],  
 ['bat', 'tab'],  
];

##### **Solution:**

```
function groupAnagramsWithEmptyStrings(strs) {
```

```

const map = new Map();

for (const str of strs) {
  const sorted = str.split("").sort().join("");
  if (!map.has(sorted)) {
    map.set(sorted, []);
  }
  map.get(sorted).push(str);
}

return Array.from(map.values());
}

// Test
console.log(
  groupAnagramsWithEmptyStrings(["", "eat", "tea", "", "bat", "tab"])
);
// Output: [[ "", "" ], [ "eat", "tea" ], [ "bat", "tab" ]]
console.log(groupAnagramsWithEmptyStrings([""]));
// Output: [ [ "" ] ]
console.log(groupAnagramsWithEmptyStrings(["a", ""]));
// Output: [ [ "" ], [ "a" ] ]

```

### Group Anagrams (With Mixed Case)

**128. Problem: Group anagrams while treating uppercase and lowercase letters as identical.**

**Example:**

Input: strs = ["Eat", "Tea", "ate", "bat", "Tab"];  
Output: [  
 ["Eat", "Tea", "ate"],  
 ["bat", "Tab"],  
];

**Solution:**

```

function groupAnagramsMixedCase(strs) {
  const map = new Map();

  for (const str of strs) {
    const sorted = str.toLowerCase().split("").sort().join("");
    if (!map.has(sorted)) {
      map.set(sorted, []);
    }
    map.get(sorted).push(str);
  }
}

```

```

    return Array.from(map.values());
}

// Test
console.log(groupAnagramsMixedCase(["Eat", "Tea", "ate", "bat", "Tab"]));
// Output: [["Eat", "Tea", "ate"], ["bat", "Tab"]]
console.log(groupAnagramsMixedCase(["a", "A"]));
// Output: [["a", "A"]]

```

### Explanation:

1. **Objective:** Group anagrams while ignoring case differences.
2. **Approach:**
  - o Convert each string to lowercase before sorting.
  - o Use the lowercase sorted string as the hash key.
3. **Steps:**
  - o Normalize each string to lowercase and sort it.
  - o Use the normalized string as a key in the hash map.
  - o Group original strings under the same key.
4. **Why it Works:** Lowercasing ensures case-insensitivity while sorting groups identical characters together.
5. **Time Complexity:**  $O(n * k \log k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.
6. **Space Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

### Group Anagrams (Case-Sensitive Approach)

**129. Problem: Group anagrams while treating uppercase and lowercase letters as distinct.**

### Example:

Input: `strs = ["Eat", "Tea", "ate", "bat", "Tab"];`  
Output: `[["Eat"], ["Tea"], ["ate"], ["bat"], ["Tab"]];`

### Solution:

```

function groupAnagramsCaseSensitive(strs) {
  const map = new Map();

  for (const str of strs) {
    const sorted = str.split("").sort().join("");
    if (!map.has(sorted)) {
      map.set(sorted, []);
    }
    map.get(sorted).push(str);
  }
}

```

```

    }

    return Array.from(map.values());
}

// Test
console.log(groupAnagramsCaseSensitive(["Eat", "Tea", "ate", "bat", "Tab"]));
// Output: [["Eat"], ["Tea"], ["ate"], ["bat"], ["Tab"]]
console.log(groupAnagramsCaseSensitive(["a", "A"]));
// Output: [["a"], ["A"]]
console.log(groupAnagramsCaseSensitive(["abc", "bca", "CBA"]));
// Output: [["abc", "bca"], ["CBA"]]

```

### Explanation:

1. **Objective:** Treat uppercase and lowercase letters as distinct when grouping anagrams.
2. **Approach:**
  - o Sort each string to create a unique key.
  - o Use the sorted string as the key in the hash map without normalizing case.
3. **Steps:**
  - o Sort each string directly without converting it to lowercase.
  - o Use the sorted string as a key in the hash map.
  - o Group strings under the same key.
4. **Why it Works:** Sorting ensures identical characters are grouped together, and case sensitivity is preserved by avoiding case normalization.
5. **Time Complexity:**  $O(n * k \log k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.
6. **Space Complexity:**  $O(n * k)$ , where  $n$  is the number of strings, and  $k$  is the average length of the strings.

### Group Anagrams (Unicode and Mixed Case Handling)

**130. Problem: Group anagrams for strings with Unicode characters, including special characters, while treating case differences as distinct.**

### Example:

```

Input: strs = ["déjà", "ajéd", "Déjà", "jaéD"];
Output: [
  ["déjà", "ajéd"],
  ["Déjà", "jaéD"],
];

```

### Solution:

```

function groupAnagramsUnicodeCaseSensitive(strs) {
  const map = new Map();

```

```

for (const str of strs) {
  const sorted = Array.from(str).sort().join("");
  if (!map.has(sorted)) {
    map.set(sorted, []);
  }
  map.get(sorted).push(str);
}

return Array.from(map.values());
}

// Test
console.log(
  groupAnagramsUnicodeCaseSensitive(["déjà", "ajéd", "Déjà", "jaéD"])
);
// Output: [["déjà", "ajéd"], ["Déjà", "jaéD"]]
console.log(groupAnagramsUnicodeCaseSensitive(["𠮷𠮶", "𠮶𠮷", "𠮷𠮶"]));
// Output: [["𠮷𠮶", "𠮶𠮷"], ["𠮶𠮷", "𠮷𠮶"]]

```

### Longest Substring Without Repeating Characters (Sliding Window Approach)

**131. Problem: Given a string s, find the length of the longest substring without repeating characters.**

**Example:**

Input: s = "abcabcbb"  
 Output: 3  
 Explanation: The answer is "abc", with the length of 3.

Input: s = "bbbbbb"  
 Output: 1  
 Explanation: The answer is "b", with the length of 1.

**Solution:**

```

function lengthOfLongestSubstring(s) {
  let left = 0;
  let maxLength = 0;
  const charSet = new Set();

  for (let right = 0; right < s.length; right++) {
    while (charSet.has(s[right])) {
      charSet.delete(s[left]);
      left++;
    }
    charSet.add(s[right]);
  }
}

```

```

        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}

// Test
console.log(lengthOfLongestSubstring("abcabcbb")); // Output: 3
console.log(lengthOfLongestSubstring("bbbbbb")); // Output: 1
console.log(lengthOfLongestSubstring("pwwkew")); // Output: 3

```

### Explanation:

1. **Objective:** Find the longest substring with unique characters.
2. **Approach:**
  - o Use a sliding window technique with a set to track unique characters in the current window.
  - o Expand the window by moving the right pointer and contract it by moving the left pointer when duplicates are found.
3. **Steps:**
  - o Traverse the string with the right pointer.
  - o If a duplicate is found, shrink the window by moving the left pointer and removing characters from the set until the duplicate is removed.
  - o Update the maximum length at each step.
4. **Why it Works:** The sliding window efficiently keeps track of unique characters while dynamically adjusting the window size.
5. **Time Complexity:**  $O(n)$  — Each character is added to and removed from the set at most once.
6. **Space Complexity:**  $O(k)$  — Space for the set, where  $k$  is the number of unique characters.

### Longest Substring Without Repeating Characters (Optimized Sliding Window Using Map)

**132. Problem: Optimize the sliding window approach by using a hash map to store the last seen index of each character.**

#### Example:

Input:  $s = "abcabcbb"$   
Output: 3  
Explanation: The answer is "abc", with the length of 3.

Input:  $s = "pwwkew"$   
Output: 3  
Explanation: The answer is "wke", with the length of 3.

### Solution:

```
function lengthOfLongestSubstringOptimized(s) {  
    let maxLength = 0;  
    const charIndexMap = new Map();  
    let left = 0;  
  
    for (let right = 0; right < s.length; right++) {  
        if (charIndexMap.has(s[right]) && charIndexMap.get(s[right]) >= left) {  
            left = charIndexMap.get(s[right]) + 1;  
        }  
        charIndexMap.set(s[right], right);  
        maxLength = Math.max(maxLength, right - left + 1);  
    }  
  
    return maxLength;  
}  
  
// Test  
console.log(lengthOfLongestSubstringOptimized("abcabcbb")); // Output: 3  
console.log(lengthOfLongestSubstringOptimized("bbbbbb")); // Output: 1  
console.log(lengthOfLongestSubstringOptimized("pwwkew")); // Output: 3
```

### Explanation:

1. **Objective:** Improve the efficiency of the sliding window approach using a hash map.
2. **Approach:**
  - o Use a hash map to store the last seen index of each character.
  - o If a duplicate is found, move the left pointer to  $\text{charIndexMap}[s[\text{right}]] + 1$  to skip the duplicate.
3. **Steps:**
  - o Traverse the string with the right pointer.
  - o If a character is found in the map and its index is within the current window ( $\geq \text{left}$ ), update the left pointer.
  - o Update the hash map with the current index of the character.
  - o Calculate the maximum length at each step.
4. **Why it Works:** The hash map allows direct access to the last seen index, avoiding redundant checks.
5. **Time Complexity:**  $O(n)$  — Each character is processed once.
6. **Space Complexity:**  $O(k)$  — Space for the hash map, where  $k$  is the number of unique characters.

### Longest Substring Without Repeating Characters (Brute Force Approach)

**133. Problem:** **Find the length of the longest substring without repeating characters using a brute-force approach.**

### Example:

Input: s = "abcabcb";

Output: 3;

Input: s = "bbbbbb";

Output: 1;

**Solution:**

```
function lengthOfLongestSubstringBruteForce(s) {  
    let maxLength = 0;  
  
    for (let i = 0; i < s.length; i++) {  
        let seen = new Set();  
        for (let j = i; j < s.length; j++) {  
            if (seen.has(s[j])) break;  
            seen.add(s[j]);  
            maxLength = Math.max(maxLength, j - i + 1);  
        }  
    }  
  
    return maxLength;  
}  
  
// Test  
console.log(lengthOfLongestSubstringBruteForce("abcabcb")); // Output: 3  
console.log(lengthOfLongestSubstringBruteForce("bbbbbb")); // Output: 1  
console.log(lengthOfLongestSubstringBruteForce("pwwkew")); // Output: 3
```

**Explanation:**

1. **Objective:** Check all possible substrings and find the longest one without repeating characters.
2. **Approach:**
  - o Use nested loops:
    - The outer loop sets the start of the substring.
    - The inner loop checks characters and stops when a duplicate is found.
  - o Use a set to track unique characters in the current substring.
3. **Steps:**
  - o For each starting position, iterate through the string and check if characters are unique.
  - o Update the maximum length if a longer valid substring is found.
4. **Why it Works:** By explicitly checking every possible substring, all valid substrings are considered.
5. **Time Complexity:**  $O(n^2)$  — The inner loop runs for every starting position.
6. **Space Complexity:**  $O(k)$  — Space for the set, where k is the number of unique characters.

**Longest Substring Without Repeating Characters (Handle Unicode Characters)**

**134. Problem:** Handle strings with Unicode characters, such as emojis or non-English alphabets, to find the longest substring without repeating characters.

**Example:**

Input: s = "𠮷𠮷𠮷𠮷𠮷𠮷"

Output: 3

Explanation: The substring "𠮷𠮷𠮷" has a length of 3.

**Solution:**

```
function lengthOfLongestSubstringUnicode(s) {  
    let left = 0;  
    let maxLength = 0;  
    const charSet = new Set();  
  
    for (let right = 0; right < s.length; right++) {  
        while (charSet.has(s[right])) {  
            charSet.delete(s[left]);  
            left++;  
        }  
        charSet.add(s[right]);  
        maxLength = Math.max(maxLength, right - left + 1);  
    }  
  
    return maxLength;  
}  
  
// Test  
console.log(lengthOfLongestSubstringUnicode("𠮷𠮷𠮷𠮷𠮷𠮷")); // Output: 3  
console.log(lengthOfLongestSubstringUnicode("𠮷𠮷𠮷𠮷𠮷𠮷𠮷𠮷")); // Output: 2
```

**Explanation:**

1. **Objective:** Find the longest substring with unique characters in strings containing Unicode characters.
2. **Approach:**
  - o Use the sliding window technique with a set to track unique characters.
  - o Adjust the window when duplicates are found.
3. **Steps:**
  - o Traverse the string and check if a character is already in the set.
  - o If it is, remove characters from the left until the duplicate is removed.
  - o Update the maximum length at each step.
4. **Why it Works:** The set efficiently tracks unique characters, and the sliding window ensures linear complexity.
5. **Time Complexity:** O(n) — Each character is added to and removed from the set at most once.

6. **Space Complexity:**  $O(k)$  — Space for the set, where  $k$  is the number of unique characters.

### **Longest Substring Without Repeating Characters (Track Substring Itself)**

**135. Problem:** **Return the longest substring itself instead of just its length.**

**Example:**

```
Input: s = "abcabcbb";
Output: "abc";
```

```
Input: s = "bbbbbb";
Output: "b";
```

**Solution:**

```
function findLongestSubstring(s) {
  let left = 0;
  let maxLength = 0;
  let longestSubstring = "";
  const charSet = new Set();

  for (let right = 0; right < s.length; right++) {
    while (charSet.has(s[right])) {
      charSet.delete(s[left]);
      left++;
    }
    charSet.add(s[right]);
    if (right - left + 1 > maxLength) {
      maxLength = right - left + 1;
      longestSubstring = s.slice(left, right + 1);
    }
  }

  return longestSubstring;
}

// Test
console.log(findLongestSubstring("abcabcbb")); // Output: "abc"
console.log(findLongestSubstring("bbbbbb")); // Output: "b"
console.log(findLongestSubstring("pwwkew")); // Output: "wke"
```

**Explanation:**

1. **Objective:** Return the actual longest substring without repeating characters, not just its length.
2. **Approach:**

- Use the sliding window technique with a set to track unique characters.
  - Update the longest substring whenever a new maximum length is found.
3. **Steps:**
- Adjust the window to ensure all characters are unique.
  - Update the longest substring if the current window length exceeds the maximum length.
4. **Why it Works:** The sliding window efficiently tracks the longest substring, and the slicing operation extracts the substring directly.
5. **Time Complexity:**  $O(n)$  — Each character is added to and removed from the set at most once.
6. **Space Complexity:**  $O(k)$  — Space for the set, where  $k$  is the number of unique characters.

### Longest Substring Without Repeating Characters (Optimized Sliding Window With Character Indices)

**136. Problem: Find the length of the longest substring without repeating characters using a character index map for efficient window updates.**

**Example:**

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
```

**Solution:**

```
function lengthOfLongestSubstringOptimizedMap(s) {
  const charIndexMap = new Map();
  let left = 0;
  let maxLength = 0;

  for (let right = 0; right < s.length; right++) {
    if (charIndexMap.has(s[right]) && charIndexMap.get(s[right]) >= left) {
      left = charIndexMap.get(s[right]) + 1;
    }
    charIndexMap.set(s[right], right);
    maxLength = Math.max(maxLength, right - left + 1);
  }

  return maxLength;
}
```

```
// Test
console.log(lengthOfLongestSubstringOptimizedMap("abcabcbb")); // Output: 3
console.log(lengthOfLongestSubstringOptimizedMap("pwwkew")); // Output: 3
console.log(lengthOfLongestSubstringOptimizedMap("bbbbbb")); // Output: 1
```

### Explanation:

1. **Objective:** Optimize the sliding window by efficiently updating the left pointer using a hash map.
2. **Approach:**
  - o Use a map to store the last seen index of each character.
  - o If a duplicate is encountered within the window, move the left pointer to the right of the last seen position of the duplicate.
  - o Calculate the length of the current window and update the maximum length.
3. **Steps:**
  - o Traverse the string with the right pointer.
  - o Update left if the current character has been seen before and its last position is within the current window.
  - o Store the current character's index in the map.
  - o Update the maximum length of the substring at each step.
4. **Why it Works:** The map ensures efficient tracking of the last seen index, reducing redundant window adjustments.
5. **Time Complexity:**  $O(n)$  — Each character is processed once.
6. **Space Complexity:**  $O(k)$  — Space for the map, where  $k$  is the number of unique characters.

### Longest Substring Without Repeating Characters (Return All Substrings)

**137. Problem: Return all substrings that have the maximum length without repeating characters.**

#### Example:

Input: s = "abcabcbb";  
Output: ["abc"];

Input: s = "pwwkew";  
Output: ["wke"];

#### Solution:

```
function findAllLongestSubstrings(s) {
  const charIndexMap = new Map();
  let left = 0;
  let maxLength = 0;
  const result = new Set();

  for (let right = 0; right < s.length; right++) {
```

```

if (charIndexMap.has(s[right]) && charIndexMap.get(s[right]) >= left) {
    left = charIndexMap.get(s[right]) + 1;
}
charIndexMap.set(s[right], right);

const currentLength = right - left + 1;
if (currentLength > maxLength) {
    maxLength = currentLength;
    result.clear();
    result.add(s.slice(left, right + 1));
} else if (currentLength === maxLength) {
    result.add(s.slice(left, right + 1));
}
}

return Array.from(result);
}

// Test
console.log(findAllLongestSubstrings("abcabcbb")); // Output: ["abc"]
console.log(findAllLongestSubstrings("pwwkew")); // Output: ["wke"]
console.log(findAllLongestSubstrings("ababcd")); // Output: ["abcd"]

```

### Explanation:

1. **Objective:** Find all substrings that have the maximum length without repeating characters.
2. **Approach:**
  - o Use the sliding window with a character index map for efficient updates.
  - o Track substrings in a set to ensure uniqueness.
  - o Update the set whenever a new maximum length is found or if the current length matches the maximum.
3. **Steps:**
  - o Traverse the string with the right pointer.
  - o Adjust left to skip duplicates using the map.
  - o Track substrings of the current maximum length in the set.
4. **Why it Works:** The set ensures unique substrings, and the map allows efficient window adjustments.
5. **Time Complexity:**  $O(n)$  — Each character is processed once.
6. **Space Complexity:**  $O(k + m)$ , where  $k$  is the number of unique characters and  $m$  is the number of substrings.

### Longest Substring Without Repeating Characters (Handle Empty Strings)

#### 138. Problem: Ensure the function handles edge cases like empty strings.

##### Example:

Input:  $s = "";$

```
Output: 0;
```

```
Input: s = " ";
```

```
Output: 1;
```

### Solution:

```
function lengthOfLongestSubstringHandleEmpty(s) {  
    if (s.length === 0) return 0;  
  
    let left = 0;  
    let maxLength = 0;  
    const charSet = new Set();  
  
    for (let right = 0; right < s.length; right++) {  
        while (charSet.has(s[right])) {  
            charSet.delete(s[left]);  
            left++;  
        }  
        charSet.add(s[right]);  
        maxLength = Math.max(maxLength, right - left + 1);  
    }  
  
    return maxLength;  
}  
  
// Test  
console.log(lengthOfLongestSubstringHandleEmpty("")); // Output: 0  
console.log(lengthOfLongestSubstringHandleEmpty(" ")); // Output: 1  
console.log(lengthOfLongestSubstringHandleEmpty("a")); // Output: 1
```

### Explanation:

1. **Objective:** Ensure the function handles edge cases like empty strings.
2. **Approach:**
  - o Return 0 immediately if the string is empty.
  - o Use the sliding window approach for all other cases.
3. **Steps:**
  - o Check if the input string is empty and return 0 if true.
  - o Use the sliding window logic for non-empty strings to find the maximum length.
4. **Why it Works:** Handling edge cases explicitly ensures the function is robust.
5. **Time Complexity:**  $O(n)$  — Each character is processed once.
6. **Space Complexity:**  $O(k)$  — Space for the set, where  $k$  is the number of unique characters.

### Longest Substring Without Repeating Characters (Two Pointers with Boolean Array)

### 139. Problem: Use a boolean array to efficiently track whether a character is already in the current substring.

#### Example:

Input: s = "abcabcb"   
Output: 3  
Explanation: The answer is "abc", with the length of 3.

Input: s = "pwwkew"   
Output: 3  
Explanation: The answer is "wke", with the length of 3.

#### Solution:

```
function lengthOfLongestSubstringBooleanArray(s) {  
    const seen = Array(128).fill(false); // ASCII size  
    let left = 0;  
    let maxLength = 0;  
  
    for (let right = 0; right < s.length; right++) {  
        while (seen[s.charCodeAt(right)]) {  
            seen[s.charCodeAt(left)] = false;  
            left++;  
        }  
        seen[s.charCodeAt(right)] = true;  
        maxLength = Math.max(maxLength, right - left + 1);  
    }  
  
    return maxLength;  
}  
  
// Test  
console.log(lengthOfLongestSubstringBooleanArray("abcabcb")); // Output: 3  
console.log(lengthOfLongestSubstringBooleanArray("pwwkew")); // Output: 3  
console.log(lengthOfLongestSubstringBooleanArray("bbbbbb")); // Output: 1
```

#### Explanation:

1. **Objective:** Use a boolean array to reduce overhead when tracking characters in the current substring.
2. **Approach:**
  - o Use an array of size 128 (ASCII size) to track whether a character is in the current substring.
  - o If a duplicate is found, move the left pointer until the duplicate is removed.
3. **Steps:**
  - o Mark characters in the boolean array as they are added to the substring.

- If a duplicate is encountered, update the boolean array while moving the left pointer.
  - Update the maximum length at each step.
4. **Why it Works:** The boolean array provides constant-time checks and updates, improving efficiency.
  5. **Time Complexity:**  $O(n)$  — Each character is processed once.
  6. **Space Complexity:**  $O(1)$  — Fixed-size array of 128 elements.

### Longest Substring Without Repeating Characters (Recursive Approach)

**140. Problem: Find the longest substring without repeating characters using recursion.**

**Example:**

```
Input: s = "abcabcbb";
Output: 3;
```

```
Input: s = "bbbbbb";
Output: 1;
```

**Solution:**

```
function lengthOfLongestSubstringRecursive(s, start = 0, seen = new Set()) {
  if (start >= s.length) return 0;

  let maxLength = 0;
  for (let i = start; i < s.length; i++) {
    if (seen.has(s[i])) break;
    seen.add(s[i]);
    maxLength = Math.max(maxLength, i - start + 1);
  }

  return Math.max(
    maxLength,
    lengthOfLongestSubstringRecursive(s, start + 1, new Set())
  );
}

// Test
console.log(lengthOfLongestSubstringRecursive("abcabcbb")); // Output: 3
console.log(lengthOfLongestSubstringRecursive("bbbbbb")); // Output: 1
console.log(lengthOfLongestSubstringRecursive("pwwkew")); // Output: 3
```

**Explanation:**

1. **Objective:** Use recursion to find the longest substring without repeating characters.
2. **Approach:**

- At each recursive step, attempt to build the longest substring starting from the current index.
  - Use a set to track unique characters in the current substring.
  - Recur for the next starting index and take the maximum result.
3. **Steps:**
- Initialize a set for the current substring.
  - Iterate from the current starting index and add characters to the set until a duplicate is found.
  - Recur for the next starting index and return the maximum length.
4. **Why it Works:** Recursion ensures all starting points are explored, and the set tracks unique characters efficiently.
5. **Time Complexity:**  $O(n^2)$  — Each starting point is processed separately.
6. **Space Complexity:**  $O(n)$  — Space for the recursive stack and the set.

### Minimum Window Substring (Sliding Window with Frequency Map)

**141. Problem:** Given two strings  $s$  and  $t$ , return the minimum window in  $s$  which will contain all the characters in  $t$ . If there is no such window, return an empty string.

**Example:**

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The substring "BANC" contains "A", "B", and "C" from `t` in the minimum window.

Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string `s` contains the single character from `t`.
```

**Solution:**

```
function minWindow(s, t) {
  if (t.length > s.length) return "";

  const tFreq = {};
  for (const char of t) {
    tFreq[char] = (tFreq[char] || 0) + 1;
  }

  let left = 0;
  let minLength = Infinity;
  let minStart = 0;
  let required = Object.keys(tFreq).length;
  let formed = 0;
```

```

const windowFreq = { };

for (let right = 0; right < s.length; right++) {
    const char = s[right];
    windowFreq[char] = (windowFreq[char] || 0) + 1;

    if (tFreq[char] && windowFreq[char] === tFreq[char]) {
        formed++;
    }
}

while (formed === required) {
    if (right - left + 1 < minLength) {
        minLength = right - left + 1;
        minStart = left;
    }

    const leftChar = s[left];
    windowFreq[leftChar]--;
    if (tFreq[leftChar] && windowFreq[leftChar] < tFreq[leftChar]) {
        formed--;
    }
    left++;
}
}

return minLength === Infinity
? ""
: s.substring(minStart, minStart + minLength);
}

// Test
console.log(minWindow("ADOBECODEBANC", "ABC")); // Output: "BANC"
console.log(minWindow("a", "a")); // Output: "a"
console.log(minWindow("a", "aa")); // Output: ""

```

### Explanation:

1. **Objective:** Find the smallest window in s that contains all characters of t.
2. **Approach:**
  - o Use the sliding window technique.
  - o Maintain a frequency map for t and a window frequency map for s.
  - o Expand the window by moving the right pointer and shrink it by moving the left pointer when the condition is met.
3. **Steps:**
  - o Traverse s with the right pointer, updating the window frequency.
  - o Check if the current window satisfies the requirements of t.
  - o If it does, update the minimum length and try to shrink the window by moving left.

4. **Why it Works:** The sliding window ensures that every possible substring is considered while minimizing redundant calculations.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k)$  — Space for the frequency maps, where m is the size of t and k is the unique characters in s.

### Minimum Window Substring (Optimized Sliding Window with Hash Map)

**142. Problem: Find the smallest substring in s that contains all the characters in t, using a more optimized hash map approach.**

**Example:**

Input: (s = "ADOBECODEBANC"), (t = "ABC");  
 Output: "BANC";

Input: (s = "a"), (t = "a");  
 Output: "a";

Input: (s = "a"), (t = "aa");  
 Output: "";

**Solution:**

```
function minWindowOptimized(s, t) {
  if (t.length > s.length) return "";

  const tFreq = new Map();
  for (const char of t) {
    tFreq.set(char, (tFreq.get(char) || 0) + 1);
  }

  let left = 0;
  let minLength = Infinity;
  let minStart = 0;
  let required = tFreq.size;
  let formed = 0;
  const windowFreq = new Map();

  for (let right = 0; right < s.length; right++) {
    const char = s[right];
    windowFreq.set(char, (windowFreq.get(char) || 0) + 1);

    if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {
      formed++;
    }

    while (formed === required) {
```

```

if (right - left + 1 < minLength) {
    minLength = right - left + 1;
    minStart = left;
}

const leftChar = s[left];
windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);
if (
    tFreq.has(leftChar) &&
    windowFreq.get(leftChar) < tFreq.get(leftChar)
) {
    formed--;
}
left++;
}
}

return minLength === Infinity
? ""
: s.substring(minStart, minStart + minLength);
}

// Test
console.log(minWindowOptimized("ADOBECODEBANC", "ABC")); // Output: "BANC"
console.log(minWindowOptimized("a", "a")); // Output: "a"
console.log(minWindowOptimized("a", "aa")); // Output: ""

```

### **Explanation:**

1. **Objective:** Optimize space by using Map to store character frequencies instead of plain objects.
2. **Approach:**
  - o Use Map to handle character frequency for t and the current window in s.
  - o Check if the window contains all required characters using the formed variable.
  - o Minimize the window size while maintaining the requirement.
3. **Steps:**
  - o Build a frequency map for t using Map.
  - o Expand the window by moving the right pointer.
  - o When the window is valid, shrink it by moving the left pointer to find the smallest substring.
4. **Why it Works:** Using Map ensures better space efficiency and allows easier handling of large character sets.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

### **Minimum Window Substring (Handling Case Sensitivity)**

### 143. Problem: Modify the function to handle case-insensitive substrings.

**Example:**

Input: (s = "AdObEcOdEbAnC"), (t = "aBc");  
Output: "bAnC";

**Solution:**

```
function minWindowCaseInsensitive(s, t) {  
    s = s.toLowerCase();  
    t = t.toLowerCase();  
  
    const tFreq = new Map();  
    for (const char of t) {  
        tFreq.set(char, (tFreq.get(char) || 0) + 1);  
    }  
  
    let left = 0;  
    let minLength = Infinity;  
    let minStart = 0;  
    let required = tFreq.size;  
    let formed = 0;  
    const windowFreq = new Map();  
  
    for (let right = 0; right < s.length; right++) {  
        const char = s[right];  
        windowFreq.set(char, (windowFreq.get(char) || 0) + 1);  
  
        if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {  
            formed++;  
        }  
  
        while (formed === required) {  
            if (right - left + 1 < minLength) {  
                minLength = right - left + 1;  
                minStart = left;  
            }  
  
            const leftChar = s[left];  
            windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);  
            if (  
                tFreq.has(leftChar) &&  
                windowFreq.get(leftChar) < tFreq.get(leftChar)  
            ) {  
                formed--;  
            }  
            left++;  
        }  
    }  
}
```

```
}

return minLength === Infinity
? ""
: s.substring(minStart, minStart + minLength);
}

// Test
console.log(minWindowCaseInsensitive("AdObEcOdEbAnC", "aBc")); // Output: "bAnC"
console.log(minWindowCaseInsensitive("a", "A")); // Output: "a"
console.log(minWindowCaseInsensitive("A", "a")); // Output: "A"
```

## Explanation:

1. **Objective:** Handle case insensitivity by converting both s and t to lowercase.
  2. **Approach:**
    - o Normalize both strings by converting to lowercase.
    - o Apply the sliding window technique using frequency maps to find the minimum substring.
  3. **Steps:**
    - o Convert s and t to lowercase before processing.
    - o Use a similar approach as in the previous solution to find the smallest substring.
  4. **Why it Works:** Normalizing both strings ensures that case differences are ignored when checking character matches.
  5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
  6. **Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

## Minimum Window Substring (With Unicode Character Support)

**144. Problem:** Modify the function to support Unicode characters, such as emojis or characters from non-Latin scripts.

## Example

Input: (s = "apple orange grapes apple orange grapes"), (t = "grapes orange");  
Output: "grapes orange";

Input: ( $s = \text{“你好世界你好”}$ ), ( $t = \text{“世界”}$ );

Output: "世界";

### Solution:

```
function minWindowUnicode(s, t) {  
    const tFreq = new Map();  
    for (const char of t) {
```

```

        tFreq.set(char, (tFreq.get(char) || 0) + 1);
    }

let left = 0;
let minLength = Infinity;
let minStart = 0;
let required = tFreq.size;
let formed = 0;
const windowFreq = new Map();

for (let right = 0; right < s.length; right++) {
    const char = s[right];
    windowFreq.set(char, (windowFreq.get(char) || 0) + 1);

    if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {
        formed++;
    }
}

while (formed === required) {
    if (right - left + 1 < minLength) {
        minLength = right - left + 1;
        minStart = left;
    }
}

const leftChar = s[left];
windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);
if (
    tFreq.has(leftChar) &&
    windowFreq.get(leftChar) < tFreq.get(leftChar)
) {
    formed--;
}
left++;
}

return minLength === Infinity
? ""
: Array.from(s)
  .slice(minStart, minStart + minLength)
  .join("");
}

// Test
console.log(minWindowUnicode("𠮷 𠮷 𠮷 𠮷 𠮷", "𠮷 𠮷")); // Output: "𠮷 𠮷"
console.log(minWindowUnicode("你好世界你好", "世界")); // Output: "世界"
console.log(minWindowUnicode("abc", "z")); // Output: ""

```

### **Explanation:**

1. **Objective:** Ensure the function supports Unicode characters.
2. **Approach:**
  - o Use Array.from to handle strings containing Unicode characters properly.
  - o Apply the sliding window technique using frequency maps for both t and s.
3. **Steps:**
  - o Create frequency maps for t and the current window in s.
  - o Expand the window with right and shrink it with left when the conditions are met.
  - o Return the smallest substring as a joined string of Unicode characters.
4. **Why it Works:** Array.from properly handles Unicode strings, which might otherwise cause issues with character indexing.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

### **Minimum Window Substring (Find All Valid Windows)**

**145. Problem:** **Return all valid windows in s that contain all characters of t and are minimal in length.**

#### **Example:**

Input: (s = "ADOBECODEBANC"), (t = "ABC");  
Output: ["BANC"];

Input: (s = "aabc"), (t = "abc");  
Output: ["abc"];

#### **Solution:**

```
function findAllMinWindows(s, t) {  
    const tFreq = new Map();  
    for (const char of t) {  
        tFreq.set(char, (tFreq.get(char) || 0) + 1);  
    }  
  
    let left = 0;  
    let minLength = Infinity;  
    const result = [];  
    let required = tFreq.size;  
    let formed = 0;  
    const windowFreq = new Map();  
  
    for (let right = 0; right < s.length; right++) {  
        const char = s[right];  
        windowFreq.set(char, (windowFreq.get(char) || 0) + 1);
```

```

if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {
    formed++;
}

while (formed === required) {
    if (right - left + 1 <= minLength) {
        if (right - left + 1 < minLength) {
            result.length = 0;
            minLength = right - left + 1;
        }
        result.push(s.slice(left, right + 1));
    }
}

const leftChar = s[left];
windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);
if (
    tFreq.has(leftChar) &&
    windowFreq.get(leftChar) < tFreq.get(leftChar)
) {
    formed--;
}
left++;
}

return result;
}

// Test
console.log(findAllMinWindows("ADOBECODEBANC", "ABC")); // Output: ["BANC"]
console.log(findAllMinWindows("aabc", "abc")); // Output: ["abc"]
console.log(findAllMinWindows("a", "aa")); // Output: []

```

### Explanation:

1. **Objective:** Return all minimal windows that satisfy the condition, not just one.
2. **Approach:**
  - o Track all windows that match the minimal length using a list.
  - o Update the list whenever a new minimal window is found.
3. **Steps:**
  - o Use the sliding window approach to find valid windows.
  - o If a new minimal window is found, clear the list and update it.
  - o If the current window matches the minimal length, add it to the list.
4. **Why it Works:** Tracking all windows ensures no valid minimal substring is missed.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k + w)$  — m for the t frequency map, k for the window map, and w for the result array.

## Minimum Window Substring (Character Order Irrelevant)

**146. Problem:** Find the minimum window in s that contains all characters in t, where the order of characters in t is irrelevant.

**Example:**

Input: (s = "ADOBECODEBANC"), (t = "CBA");  
Output: "BANC";

Input: (s = "a"), (t = "a");  
Output: "a";

Input: (s = "a"), (t = "aa");  
Output: "";

**Solution:**

```
function minWindowUnordered(s, t) {  
    const tFreq = new Map();  
    for (const char of t) {  
        tFreq.set(char, (tFreq.get(char) || 0) + 1);  
    }  
  
    let left = 0;  
    let minLength = Infinity;  
    let minStart = 0;  
    const windowFreq = new Map();  
    let required = tFreq.size;  
    let formed = 0;  
  
    for (let right = 0; right < s.length; right++) {  
        const char = s[right];  
        windowFreq.set(char, (windowFreq.get(char) || 0) + 1);  
  
        if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {  
            formed++;  
        }  
  
        while (formed === required) {  
            if (right - left + 1 < minLength) {  
                minLength = right - left + 1;  
                minStart = left;  
            }  
  
            const leftChar = s[left];  
            windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);  
            if (  
                tFreq.has(leftChar) &&  
                windowFreq.get(leftChar) < tFreq.get(leftChar)
```

```

        )
    formed--;
}
left++;
}
}

return minLength === Infinity
? ""
: s.substring(minStart, minStart + minLength);
}

// Test
console.log(minWindowUnordered("ADOBECODEBANC", "CBA")); // Output: "BANC"
console.log(minWindowUnordered("a", "a")); // Output: "a"
console.log(minWindowUnordered("a", "aa")); // Output: ""

```

### Explanation:

1. **Objective:** Ensure the minimum window substring includes all characters of t regardless of their order.
2. **Approach:**
  - o Use a frequency map to track occurrences of each character in t.
  - o Use a sliding window to track occurrences in s and adjust as needed.
3. **Steps:**
  - o Traverse s with the right pointer to expand the window.
  - o Check if the window meets the condition by matching all characters from t.
  - o If valid, shrink the window by moving the left pointer and updating the result.
4. **Why it Works:** The frequency map ensures the characters and their counts match, regardless of order.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

### Minimum Window Substring (Find Window with Extra Characters Allowed)

**147. Problem: Modify the function to allow extra characters in the window that are not in t.**

#### Example:

Input: (s = "ADOBECODEBANCXYZ"), (t = "ABC");  
Output: "BANC";

Input: (s = "aa"), (t = "a");  
Output: "a";

**Solution:**

```
function minWindowWithExtraChars(s, t) {
    const tFreq = new Map();
    for (const char of t) {
        tFreq.set(char, (tFreq.get(char) || 0) + 1);
    }

    let left = 0;
    let minLength = Infinity;
    let minStart = 0;
    const windowFreq = new Map();
    let required = tFreq.size;
    let formed = 0;

    for (let right = 0; right < s.length; right++) {
        const char = s[right];
        windowFreq.set(char, (windowFreq.get(char) || 0) + 1);

        if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {
            formed++;
        }

        while (formed === required) {
            if (right - left + 1 < minLength) {
                minLength = right - left + 1;
                minStart = left;
            }

            const leftChar = s[left];
            windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);
            if (
                tFreq.has(leftChar) &&
                windowFreq.get(leftChar) < tFreq.get(leftChar)
            ) {
                formed--;
            }
            left++;
        }
    }

    return minLength === Infinity
        ? ""
        : s.substring(minStart, minStart + minLength);
}

// Test
console.log(minWindowWithExtraChars("ADOBECODEBANCXYZ", "ABC")); // Output:
// "BANC"
console.log(minWindowWithExtraChars("aa", "a")); // Output: "a"
```

```
console.log(minWindowWithExtraChars("a", "aa")); // Output: ""
```

### Explanation:

1. **Objective:** Find the minimum window in s that contains all characters of t while allowing additional characters in the window.
2. **Approach:**
  - o Use the sliding window technique and track characters from t only.
  - o Ignore extra characters when counting matches.
3. **Steps:**
  - o Traverse s with the right pointer, updating the window frequency map.
  - o Only consider characters in t when updating the formed count.
  - o Minimize the window by moving the left pointer while the condition is met.
4. **Why it Works:** By focusing only on characters in t, the algorithm allows the window to include extra characters.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

### Minimum Window Substring (Allow Repeated Characters in t)

**148. Problem: Modify the function to handle cases where t contains repeated characters, and the minimum window in s must account for all occurrences of each character in t.**

### Example:

```
Input: (s = "AAADDOBECODEBANCC"), (t = "AABC");
Output: "ADOBECODEBA";
```

```
Input: (s = "a"), (t = "aa");
Output: "";
```

### Solution:

```
function minWindowWithRepeats(s, t) {
  const tFreq = new Map();
  for (const char of t) {
    tFreq.set(char, (tFreq.get(char) || 0) + 1);
  }

  let left = 0;
  let minLength = Infinity;
  let minStart = 0;
  let required = tFreq.size;
  let formed = 0;
  const windowFreq = new Map();
```

```

for (let right = 0; right < s.length; right++) {
    const char = s[right];
    windowFreq.set(char, (windowFreq.get(char) || 0) + 1);

    if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {
        formed++;
    }

    while (formed === required) {
        if (right - left + 1 < minLength) {
            minLength = right - left + 1;
            minStart = left;
        }

        const leftChar = s[left];
        windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);
        if (
            tFreq.has(leftChar) &&
            windowFreq.get(leftChar) < tFreq.get(leftChar)
        ) {
            formed--;
        }
        left++;
    }
}

return minLength === Infinity
? ""
: s.substring(minStart, minStart + minLength);
}

// Test
console.log(minWindowWithRepeats("AAADOBECODEBANCC", "AABC")); // Output:
"ADOBECODEBA"
console.log(minWindowWithRepeats("a", "aa")); // Output: ""
console.log(minWindowWithRepeats("ADOBECODEBANC", "AABC")); // Output:
"ADOBECODEBA"

```

### Explanation:

1. **Objective:** Ensure the window contains all characters of t, including their repeated occurrences.
2. **Approach:**
  - o Use a frequency map to track the number of occurrences required for each character in t.
  - o Maintain a second map for the window frequency and compare it with tFreq.
3. **Steps:**
  - o Traverse s with the right pointer and update the window frequency map.
  - o Check if all characters meet the required frequency using the formed variable.

- Shrink the window by moving the left pointer, updating the result if a smaller valid window is found.
- Why it Works:** The frequency map ensures that all required counts are considered, including repeated characters.
  - Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
  - Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

### **Minimum Window Substring (Return Both Window and Length)**

**149. Problem: Modify the function to return both the minimum window substring and its length.**

**Example:**

Input: s = "ADOBECODEBANC", t = "ABC"

Output: { substring: "BANC", length: 4 }

Input: s = "a", t = "a"

Output: { substring: "a", length: 1 }

Input: s = "a", t = "aa"

Output: { substring: "", length: 0 }

**Solution:**

```
function minWindowWithLength(s, t) {
  const tFreq = new Map();
  for (const char of t) {
    tFreq.set(char, (tFreq.get(char) || 0) + 1);
  }

  let left = 0;
  let minLength = Infinity;
  let minStart = 0;
  const windowFreq = new Map();
  let required = tFreq.size;
  let formed = 0;

  for (let right = 0; right < s.length; right++) {
    const char = s[right];
    windowFreq.set(char, (windowFreq.get(char) || 0) + 1);

    if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {
      formed++;
    }

    while (formed === required) {
```

```

if (right - left + 1 < minLength) {
    minLength = right - left + 1;
    minStart = left;
}

const leftChar = s[left];
windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);
if (
    tFreq.has(leftChar) &&
    windowFreq.get(leftChar) < tFreq.get(leftChar)
) {
    formed--;
}
left++;
}
}

if (minLength === Infinity) return { substring: "", length: 0 };

return {
    substring: s.substring(minStart, minStart + minLength),
    length: minLength,
};

// Test
console.log(minWindowWithLength("ADOBECODEBANC", "ABC")); // Output: {
    substring: "BANC", length: 4 }
console.log(minWindowWithLength("a", "a")); // Output: { substring: "a", length: 1 }
console.log(minWindowWithLength("a", "aa")); // Output: { substring: "", length: 0 }

```

### Explanation:

1. **Objective:** Return both the substring and its length for better clarity and usability.
2. **Approach:**
  - o Use the sliding window technique to find the smallest valid substring.
  - o Track the minimum length and start index of the substring.
  - o Return the substring and its length in an object.
3. **Steps:**
  - o Traverse s with the right pointer, updating the window frequency map.
  - o Shrink the window with the left pointer to find the minimum substring.
  - o Return the substring and its length after traversing s.
4. **Why it Works:** Adding the length as part of the output provides additional information without affecting the algorithm.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

## Minimum Window Substring (Optimized Return of Both Substring and Length)

**150. Problem:** Optimize the solution to find and return both the minimum window substring and its length efficiently.

**Example:**

Input: s = "ADOBECODEBANC", t = "ABC"

Output: { substring: "BANC", length: 4 }

Input: s = "a", t = "a"

Output: { substring: "a", length: 1 }

Input: s = "a", t = "aa"

Output: { substring: "", length: 0 }

**Solution:**

```
function minWindowSubstringAndLength(s, t) {
    if (t.length > s.length) return { substring: "", length: 0 };

    const tFreq = new Map();
    for (const char of t) {
        tFreq.set(char, (tFreq.get(char) || 0) + 1);
    }

    const windowFreq = new Map();
    let left = 0;
    let minLength = Infinity;
    let minStart = 0;
    let required = tFreq.size;
    let formed = 0;

    for (let right = 0; right < s.length; right++) {
        const char = s[right];
        windowFreq.set(char, (windowFreq.get(char) || 0) + 1);

        if (tFreq.has(char) && windowFreq.get(char) === tFreq.get(char)) {
            formed++;
        }

        while (formed === required) {
            if (right - left + 1 < minLength) {
                minLength = right - left + 1;
                minStart = left;
            }
        }

        const leftChar = s[left];
        windowFreq.set(leftChar, windowFreq.get(leftChar) - 1);
        if (

```

```

        tFreq.has(leftChar) &&
        windowFreq.get(leftChar) < tFreq.get(leftChar)
    ) {
        formed--;
    }
    left++;
}
}

return minLength === Infinity
? { substring: "", length: 0 }
: {
    substring: s.substring(minStart, minStart + minLength),
    length: minLength,
};
}
}

// Test
console.log(minWindowSubstringAndLength("ADOBECODEBANC", "ABC")); // Output: {
substring: "BANC", length: 4 }
console.log(minWindowSubstringAndLength("a", "a")); // Output: { substring: "a", length: 1 }
console.log(minWindowSubstringAndLength("a", "aa")); // Output: { substring: "", length: 0 }
}

```

### **Explanation:**

1. **Objective:** Ensure both the substring and its length are returned efficiently.
2. **Approach:**
  - o Use sliding window and hash maps to efficiently track character counts in t and the current window in s.
  - o Minimize the window size by moving the left pointer while ensuring all required characters are included.
3. **Steps:**
  - o Traverse s with the right pointer and update the window frequency.
  - o Check if the current window satisfies the requirement and shrink it with left.
  - o Update the result with the smallest substring and its length.
4. **Why it Works:** The use of hash maps ensures efficient lookups, and maintaining the minLength ensures minimal space.
5. **Time Complexity:**  $O(n + m)$  — n for traversing s, m for building the t frequency map.
6. **Space Complexity:**  $O(m + k)$  — m for the t frequency map, and k for the window map.

### **Valid Parentheses (Basic Stack Approach)**

**151. Problem:** Given a string s containing just the characters (, ), {, }, [ and ], determine if the input string is valid.

A string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

### Example:

Input: s = "()";

Output: true;

Input: s = "()[]{}";

Output: true;

Input: s = "()[";

Output: false;

### Solution:

```
function isValidParentheses(s) {
  const stack = [];
  const matchingBrackets = {
    ")": "(",
    "}": "{",
    "]": "[",
  };

  for (const char of s) {
    if (char === "(" || char === "{" || char === "[") {
      stack.push(char);
    } else if (
      matchingBrackets[char] &&
      stack.pop() !== matchingBrackets[char]
    ) {
      return false;
    }
  }

  return stack.length === 0;
}

// Test
console.log(isValidParentheses("()")); // Output: true
console.log(isValidParentheses("()[]{}")); // Output: true
console.log(isValidParentheses("()[")); // Output: false
console.log(isValidParentheses("{[()]}")); // Output: true
console.log(isValidParentheses("{[()]}")); // Output: false
```

### Explanation:

1. **Objective:** Check if the string contains valid and balanced parentheses.

## 2. Approach:

- Use a stack to track open brackets.
- Push open brackets onto the stack and pop when encountering a closing bracket.
- Ensure the popped bracket matches the current closing bracket.

## 3. Steps:

- Traverse the string character by character.
- Push open brackets onto the stack.
- For closing brackets, check if the top of the stack matches its corresponding opening bracket. If not, return false.
- After traversal, ensure the stack is empty.

4. Why it Works: The stack maintains the order of nested brackets, ensuring proper matching.

5. Time Complexity: O(n) — Traverse the string once.

6. Space Complexity: O(n) — Space used by the stack.

## Valid Parentheses (Support for Additional Characters)

**152. Problem: Validate if a string containing parentheses and other characters (e.g., letters, numbers) is valid. The non-parenthesis characters should be ignored.**

**Example:**

Input: s = "a(b)c";  
Output: true;

Input: s = "[{a+b}\*(x/y)]";  
Output: true;

Input: s = "(a+b]\*c";  
Output: false;

**Solution:**

```
function isValidParenthesesWithExtras(s) {  
    const stack = [];  
    const matchingBrackets = {  
        ")": "(",  
        "}": "{",  
        "]": "[",  
    };  
  
    for (const char of s) {  
        if (char === "(" || char === "{" || char === "[") {  
            stack.push(char);  
        } else if (char === ")" || char === "}" || char === "]") {  
            if (stack.pop() !== matchingBrackets[char]) {  
                return false;  
            }
    }
}
```

```

    }
}

return stack.length === 0;
}

// Test
console.log(isValidParenthesesWithExtras("a(b)c")); // Output: true
console.log(isValidParenthesesWithExtras("{ [a+b]*(x/y)}")); // Output: true
console.log(isValidParenthesesWithExtras("(a+b]*c")); // Output: false
console.log(isValidParenthesesWithExtras("a+b")); // Output: true
console.log(isValidParenthesesWithExtras("{[()]}")); // Output: true

```

### Explanation:

1. **Objective:** Validate parentheses while ignoring non-parenthesis characters.
2. **Approach:**
  - o Traverse the string, pushing open brackets onto a stack.
  - o For closing brackets, check if the stack's top matches the expected opening bracket.
  - o Ignore characters that are neither opening nor closing brackets.
3. **Steps:**
  - o Maintain a stack for open brackets.
  - o Push open brackets onto the stack.
  - o For closing brackets, pop from the stack and validate.
  - o Ignore non-bracket characters.
4. **Why it Works:** By ignoring irrelevant characters, only the brackets' validity is evaluated.
5. **Time Complexity:** O(n) — Traverse the string once.
6. **Space Complexity:** O(n) — Space used by the stack.

### Valid Parentheses (Early Exit Optimization)

**153. Problem: Optimize the validation to return false as soon as the number of closing brackets exceeds the number of opening brackets.**

#### Example:

Input: s = "D]";  
Output: false;

Input: s = "{[()]}";  
Output: true;

Input: s = "]";  
Output: false;

### Solution:

```
function isValidParenthesesEarlyExit(s) {  
    const stack = [];  
    const matchingBrackets = {  
        ")": "(",  
        "}": "{",  
        "]": "["  
    };  
  
    for (const char of s) {  
        if (char === "(" || char === "{" || char === "[") {  
            stack.push(char);  
        } else if (char === ")" || char === "}" || char === "]") {  
            if (stack.length === 0 || stack.pop() !== matchingBrackets[char]) {  
                return false;  
            }  
        }  
    }  
  
    return stack.length === 0;  
}  
  
// Test  
console.log(isValidParenthesesEarlyExit("([)]")); // Output: false  
console.log(isValidParenthesesEarlyExit("{[()]}")); // Output: true  
console.log(isValidParenthesesEarlyExit("]")); // Output: false  
console.log(isValidParenthesesEarlyExit("{[{}]}")); // Output: false
```

### Explanation:

1. **Objective:** Stop processing the string as soon as the parentheses become invalid.
2. **Approach:**
  - o Use a stack to track opening brackets.
  - o Immediately return false if a closing bracket has no matching opening bracket in the stack.
3. **Steps:**
  - o Traverse the string.
  - o Push open brackets onto the stack.
  - o For closing brackets, pop the stack and validate immediately. If invalid, return false.
  - o After traversal, ensure the stack is empty.
4. **Why it Works:** Early exit minimizes unnecessary processing when the string is already invalid.
5. **Time Complexity:**  $O(n)$  — Traverse the string once.
6. **Space Complexity:**  $O(n)$  — Space used by the stack.

### Valid Parentheses (Handle Only One Type of Bracket)

**154. Problem: Modify the function to validate strings with only one type of bracket, e.g., 0.**

**Example:**

Input: s = "((())";

Output: true;

Input: s = "(()";

Output: false;

Input: s = ")(";

Output: false;

**Solution:**

```
function isValidSingleType(s) {  
    let count = 0;  
  
    for (const char of s) {  
        if (char === "(") {  
            count++;  
        } else if (char === ")") {  
            count--;  
            if (count < 0) return false; // More closing brackets than opening  
        }  
    }  
  
    return count === 0; // All brackets are matched  
}  
  
// Test  
console.log(isValidSingleType("((())")); // Output: true  
console.log(isValidSingleType("(()")); // Output: false  
console.log(isValidSingleType(")(")); // Output: false  
console.log(isValidSingleType("()()")); // Output: true
```

**Explanation:**

1. **Objective:** Validate strings containing only one type of bracket.
2. **Approach:**
  - o Use a counter instead of a stack to track the balance between opening and closing brackets.
  - o Increment for ( and decrement for ).
  - o Return false immediately if the count goes negative (more closing brackets than opening).
3. **Steps:**
  - o Traverse the string, updating the count for each bracket.

- If the count becomes negative, return false.
  - After traversal, ensure the count is zero (balanced brackets).
- Why it Works:** The counter efficiently tracks the balance without requiring a stack.
  - Time Complexity:** O(n) — Traverse the string once.
  - Space Complexity:** O(1) — Constant space for the counter.

### **Valid Parentheses (Support for Nested Parentheses)**

**155. Problem: Validate a string with nested parentheses and ensure they are properly closed in the correct order.**

**Example:**

Input: s = "[()]";  
Output: true;

Input: s = "[()]";  
Output: false;

Input: s = "(({{[]}}))";  
Output: true;

**Solution:**

```
function isValidNestedParentheses(s) {
  const stack = [];
  const matchingBrackets = {
    ")": "(",
    "}": "{",
    "]": "[",
  };

  for (const char of s) {
    if (char === "(" || char === "{" || char === "[") {
      stack.push(char);
    } else if (char === ")" || char === "}" || char === "]") {
      if (stack.pop() !== matchingBrackets[char]) {
        return false;
      }
    }
  }

  return stack.length === 0;
}

// Test
console.log(isValidNestedParentheses("{[()]}")); // Output: true
console.log(isValidNestedParentheses("{[()]}")); // Output: false
console.log(isValidNestedParentheses("(({{[]}}))")); // Output: true
```

```
console.log(isValidNestedParentheses("{[]}")); // Output: false
console.log(isValidNestedParentheses("")); // Output: true
```

### Explanation:

1. **Objective:** Ensure that nested parentheses are closed in the correct order.
2. **Approach:**
  - o Use a stack to track open brackets.
  - o Push open brackets onto the stack and pop them when encountering a matching closing bracket.
  - o Return false immediately if mismatched brackets are found.
3. **Steps:**
  - o Traverse the string character by character.
  - o Push open brackets onto the stack.
  - o For closing brackets, pop the stack and validate that the popped bracket matches.
  - o After traversal, ensure the stack is empty.
4. **Why it Works:** The stack ensures proper handling of nested structures by maintaining order.
5. **Time Complexity:** O(n) — Traverse the string once.
6. **Space Complexity:** O(n) — Space used by the stack.

### Valid Parentheses (Longest Valid Parentheses Substring)

#### 156. Problem: Find the length of the longest substring with valid parentheses in the given string.

##### Example:

```
Input: s = "()"  
Output: 2  
Explanation: The longest valid substring is "()".
```

```
Input: s = ")()())"  
Output: 4  
Explanation: The longest valid substring is "()()".
```

```
Input: s = ""  
Output: 0
```

##### Solution:

```
function longestValidParentheses(s) {
  const stack = [-1]; // Initialize stack with -1 to handle edge cases
  let maxLength = 0;

  for (let i = 0; i < s.length; i++) {
    if (s[i] === "(") {
```

```

        stack.push(i);
    } else {
        stack.pop();
        if (stack.length === 0) {
            stack.push(i);
        } else {
            maxLength = Math.max(maxLength, i - stack[stack.length - 1]);
        }
    }

    return maxLength;
}

// Test
console.log(longestValidParentheses("()")); // Output: 2
console.log(longestValidParentheses("()()")); // Output: 4
console.log(longestValidParentheses ""); // Output: 0
console.log(longestValidParentheses "(()")); // Output: 2

```

### Explanation:

1. **Objective:** Find the length of the longest valid substring of parentheses.
2. **Approach:**
  - o Use a stack to store indices of unmatched parentheses.
  - o Push the index of every unmatched ( and pop for every matched ).
  - o Track the length of valid substrings by comparing the current index with the top of the stack.
3. **Steps:**
  - o Push -1 onto the stack initially to handle edge cases.
  - o Traverse the string, pushing indices of ( and popping for ).
  - o Update the maximum length for every valid substring found.
4. **Why it Works:** The stack keeps track of unmatched parentheses, allowing efficient calculation of valid substrings.
5. **Time Complexity:** O(n) — Traverse the string once.
6. **Space Complexity:** O(n) — Space used by the stack.

### Valid Parentheses (Check for Balanced Parentheses Without Nesting)

#### 157. Problem: Check if the string has balanced parentheses but no nested structures.

##### Example:

Input: s = "()";  
Output: true;

Input: s = "(())";  
Output: false;

```
Input: s = "((())";  
Output: false;
```

### Solution:

```
function isBalancedWithoutNesting(s) {  
    let balance = 0;  
  
    for (const char of s) {  
        if (char === "(") {  
            if (balance > 0) return false; // Nested structure detected  
            balance++;  
        } else if (char === ")") {  
            balance--;  
            if (balance < 0) return false; // Unbalanced closing bracket  
        }  
    }  
  
    return balance === 0;  
}  
  
// Test  
console.log(isBalancedWithoutNesting("()")); // Output: true  
console.log(isBalancedWithoutNesting("(()")); // Output: false  
console.log(isBalancedWithoutNesting("((())")); // Output: false  
console.log(isBalancedWithoutNesting("")); // Output: true
```

### Explanation:

1. **Objective:** Ensure the string has balanced parentheses without nested structures.
2. **Approach:**
  - o Use a balance counter to track the number of open parentheses.
  - o Detect nested structures by checking if the balance counter exceeds 0 after encountering an open parenthesis.
3. **Steps:**
  - o Traverse the string, incrementing the balance counter for ( and decrementing for ).
  - o Return false if the balance counter exceeds 0 when processing ( or goes negative for ).
  - o Ensure the balance counter is 0 after traversal.
4. **Why it Works:** By disallowing positive balance after (, nesting is detected and rejected.
5. **Time Complexity:** O(n) — Traverse the string once.
6. **Space Complexity:** O(1) — Constant space used for the balance counter.

### Valid Parentheses (Validate with Different Sets of Brackets)

**158. Problem: Validate parentheses for multiple bracket types, e.g., <> in addition to (), {}, and [].**

**Example:**

Input: s = "<{[()]}>";

Output: true;

Input: s = "<{[()]}>";

Output: false;

Input: s = "<()>";

Output: true;

**Solution:**

```
function isValidWithAdditionalBrackets(s) {  
    const stack = [];  
    const matchingBrackets = {  
        ")": "(",  
        "}": "{",  
        "]": "[",  
        ">": "<",  
    };  
  
    for (const char of s) {  
        if (["(", "{", "[", "<"].includes(char)) {  
            stack.push(char);  
        } else if (")", "}", "]", ">"].includes(char)) {  
            if (stack.pop() !== matchingBrackets[char]) {  
                return false;  
            }
        }
    }  
  
    return stack.length === 0;
}  
  
// Test  
console.log(isValidWithAdditionalBrackets("<{[()]}>")); // Output: true  
console.log(isValidWithAdditionalBrackets("<{[()]}>")); // Output: false  
console.log(isValidWithAdditionalBrackets("<()>")); // Output: true  
console.log(isValidWithAdditionalBrackets("<{}>")); // Output: false  
console.log(isValidWithAdditionalBrackets("")); // Output: true
```

**Explanation:**

- Objective:** Extend the validation function to support additional types of brackets.

## 2. Approach:

- o Add < and > to the list of valid brackets.
- o Update the matching brackets map to include < and >.

## 3. Steps:

- o Push all opening brackets ((, {, [, <) onto the stack.
- o Pop and validate when encountering closing brackets (), }, ], >).
- o After traversal, ensure the stack is empty.

4. Why it Works: The stack tracks the order of open brackets, ensuring proper closure and matching for all types of brackets.

5. Time Complexity: O(n) — Traverse the string once.

6. Space Complexity: O(n) — Space used by the stack.

## Valid Parentheses (Balanced Without Stack)

**159. Problem: Determine if parentheses are balanced without using a stack, specifically for strings containing only ( and ).**

### Example:

```
Input: s = "()";  
Output: true;
```

```
Input: s = "(()";  
Output: false;
```

```
Input: s = ")()(";  
Output: false;
```

### Solution:

```
function isBalancedWithoutStack(s) {  
    let balance = 0;  
  
    for (const char of s) {  
        if (char === "(") {  
            balance++;  
        } else if (char === ")") {  
            balance--;  
            if (balance < 0) return false; // More closing brackets than opening  
        }  
    }  
  
    return balance === 0; // Ensure no unmatched opening brackets remain  
}  
  
// Test  
console.log(isBalancedWithoutStack("()")); // Output: true  
console.log(isBalancedWithoutStack("(()")); // Output: false  
console.log(isBalancedWithoutStack(")()(")); // Output: false
```

```
console.log(isBalancedWithoutStack("()")); // Output: true  
console.log(isBalancedWithoutStack("")); // Output: true
```

### Explanation:

1. **Objective:** Simplify the validation for strings with only ( and ) using a counter instead of a stack.
2. **Approach:**
  - o Use a balance counter to track the net number of open parentheses.
  - o Increment for ( and decrement for ); return false if the counter goes negative.
3. **Steps:**
  - o Traverse the string and adjust the balance counter for each bracket.
  - o Ensure the counter is 0 at the end to confirm all brackets are matched.
4. **Why it Works:** The counter tracks unmatched opening brackets, ensuring proper balancing without needing a stack.
5. **Time Complexity:** O(n) — Traverse the string once.
6. **Space Complexity:** O(1) — Constant space for the counter.

### Valid Parentheses (With Maximum Depth)

#### 160. Problem: Find the maximum depth of nested parentheses in the string.

##### Example:

Input: s = "(1+(2\*3)+((8)/4))+1";  
Output: 3;

Input: s = "(1)+((2))+(((3)))";  
Output: 3;

Input: s = "1+(2\*3)/(2-1)";  
Output: 1;

##### Solution:

```
function maxDepth(s) {  
    let maxDepth = 0;  
    let currentDepth = 0;  
  
    for (const char of s) {  
        if (char === "(") {  
            currentDepth++;  
            maxDepth = Math.max(maxDepth, currentDepth);  
        } else if (char === ")") {  
            currentDepth--;  
            if (currentDepth < 0) return -1; // Invalid if closing bracket has no match  
        }  
    }  
}
```

```

        return currentDepth === 0 ? maxDepth : -1; // Return -1 if unbalanced
    }

// Test
console.log(maxDepth("(1+(2*3)+((8)/4))+1")); // Output: 3
console.log(maxDepth("(1)+((2))+(((3)))")); // Output: 3
console.log(maxDepth("1+(2*3)/(2-1)")); // Output: 1
console.log(maxDepth("()")); // Output: -1 (unbalanced)
console.log(maxDepth("")); // Output: 0

```

### **Explanation:**

1. **Objective:** Determine the maximum level of nested parentheses in the string.
2. **Approach:**
  - o Use a currentDepth counter to track the current level of nesting.
  - o Update the maxDepth whenever currentDepth increases.
  - o Decrement currentDepth for each closing bracket and return -1 if it goes negative.
3. **Steps:**
  - o Traverse the string, adjusting currentDepth for each ( and ).
  - o Track the maximum value of currentDepth.
  - o Ensure the currentDepth is 0 at the end to confirm the parentheses are balanced.
4. **Why it Works:** The counters efficiently track nesting levels and detect imbalances.
5. **Time Complexity:** O(n) — Traverse the string once.
6. **Space Complexity:** O(1) — Constant space for the counters.

### **Valid Palindrome (Basic Case-Insensitive Check)**

**161. Problem:** Given a string s, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

#### **Example:**

Input: s = "A man, a plan, a canal: Panama";  
Output: true;

Input: s = "race a car";  
Output: false;

#### **Solution:**

```

function isPalindrome(s) {
    const cleaned = s.replace(/\[^a-zA-Z0-9]/g, "").toLowerCase();
    let left = 0;

```

```

let right = cleaned.length - 1;

while (left < right) {
    if (cleaned[left] !== cleaned[right]) {
        return false;
    }
    left++;
    right--;
}

return true;
}

// Test
console.log(isPalindrome("A man, a plan, a canal: Panama")); // Output: true
console.log(isPalindrome("race a car")); // Output: false
console.log(isPalindrome(" ")); // Output: true
console.log(isPalindrome("OP")); // Output: false

```

### **Explanation:**

1. **Objective:** Check if a string is a palindrome after removing non-alphanumeric characters and ignoring cases.
2. **Approach:**
  - o Clean the string by removing all non-alphanumeric characters using a regular expression.
  - o Convert the cleaned string to lowercase to handle case insensitivity.
  - o Use two pointers (left and right) to compare characters from the start and end of the string.
3. **Steps:**
  - o Clean the input string and initialize pointers.
  - o Move the pointers inward while comparing the characters.
  - o If a mismatch is found, return false. If pointers meet without mismatches, return true.
4. **Why it Works:** By only considering alphanumeric characters and comparing symmetrically, the function efficiently determines if the string is a palindrome.
5. **Time Complexity:**  $O(n)$  — Single traversal of the string after cleaning.
6. **Space Complexity:**  $O(n)$  — Space for the cleaned string.

### **Valid Palindrome (Two-Pointer Without String Cleaning)**

**162. Problem: Validate a palindrome without creating a new string by ignoring non-alphanumeric characters and cases directly during traversal.**

### **Example:**

Input: s = "A man, a plan, a canal: Panama";  
Output: **true**;

Input: s = "race a car";

Output: false;

### Solution:

```
function isPalindromeWithoutCleaning(s) {  
    let left = 0;  
    let right = s.length - 1;  
  
    while (left < right) {  
        // Skip non-alphanumeric characters  
        while (left < right && !isAlphanumeric(s[left]))) {  
            left++;  
        }  
        while (left < right && !isAlphanumeric(s[right])) {  
            right--;  
        }  
  
        // Compare the characters  
        if (s[left].toLowerCase() !== s[right].toLowerCase()) {  
            return false;  
        }  
        left++;  
        right--;  
    }  
  
    return true;  
}  
  
function isAlphanumeric(char) {  
    return /^[a-zA-Z0-9]$/.test(char);  
}  
  
// Test  
console.log(isPalindromeWithoutCleaning("A man, a plan, a canal: Panama")); // Output:  
true  
console.log(isPalindromeWithoutCleaning("race a car")); // Output: false  
console.log(isPalindromeWithoutCleaning(" ")); // Output: true  
console.log(isPalindromeWithoutCleaning("OP")); // Output: false
```

### Explanation:

1. **Objective:** Optimize palindrome checking by avoiding additional string processing.
2. **Approach:**
  - o Use two pointers (left and right) to traverse the string from both ends.
  - o Skip non-alphanumeric characters using a helper function (isAlphanumeric).
  - o Compare characters at the left and right pointers, ignoring case.
3. **Steps:**

- Initialize two pointers at the start and end of the string.
  - Skip invalid characters by moving the pointers inward.
  - Compare characters; return false if they don't match.
  - If the loop completes without mismatches, return true.
4. **Why it Works:** By directly validating characters during traversal, this approach avoids creating a new string, saving memory.
  5. **Time Complexity:**  $O(n)$  — Single traversal of the string.
  6. **Space Complexity:**  $O(1)$  — No additional space is used.

### Valid Palindrome (Check for Numbers Only)

**163. Problem:** Validate a string containing only numbers and check if it forms a palindrome.

**Example:**

Input: s = "12321";  
Output: true;

Input: s = "123456";  
Output: false;

**Solution:**

```
function isNumericPalindrome(s) {
  let left = 0;
  let right = s.length - 1;

  while (left < right) {
    if (s[left] !== s[right]) {
      return false;
    }
    left++;
    right--;
  }

  return true;
}

// Test
console.log(isNumericPalindrome("12321")); // Output: true
console.log(isNumericPalindrome("123456")); // Output: false
console.log(isNumericPalindrome("1221")); // Output: true
console.log(isNumericPalindrome("1")); // Output: true
```

**Explanation:**

1. **Objective:** Check if a numeric string is a palindrome.

## 2. Approach:

- o Use two pointers to compare digits from the start and end of the string.
- o Move the pointers inward after each comparison.
- o Return false if mismatched digits are found.

## 3. Steps:

- o Initialize two pointers at the start and end of the string.
- o Compare the digits at both pointers.
- o If mismatched, return false. Otherwise, continue until pointers meet.
- o If no mismatches are found, return true.

4. Why it Works: The symmetric property of palindromes ensures that only digits need to be compared.

5. Time Complexity:  $O(n)$  — Single traversal of the string.

6. Space Complexity:  $O(1)$  — No additional space is used.

## Valid Palindrome (Support for Unicode Characters)

**164. Problem: Check if a string with Unicode characters is a palindrome, considering only alphanumeric characters.**

### Example:

Input: s = "A man, a plan, a canal: Panamá";  
Output: true;

Input: s = "mañana";  
Output: false;

### Solution:

```
function isUnicodePalindrome(s) {  
    let left = 0;  
    let right = s.length - 1;  
  
    while (left < right) {  
        // Skip non-alphanumeric characters  
        while (left < right && !isAlphanumericUnicode(s[left])) {  
            left++;  
        }  
        while (left < right && !isAlphanumericUnicode(s[right])) {  
            right--;  
        }  
  
        // Compare the characters case-insensitively  
        if (s[left].toLowerCase() !== s[right].toLowerCase()) {  
            return false;  
        }  
        left++;  
        right--;  
    }  
}
```

```

    return true;
}

function isAlphanumericUnicode(char) {
  return /\p{L}|\p{N}/u.test(char); // Matches any letter or number in Unicode
}

// Test
console.log(isUnicodePalindrome("A man, a plan, a canal: Panamá")); // Output: true
console.log(isUnicodePalindrome("mañana")); // Output: false
console.log(isUnicodePalindrome("0ññ0")); // Output: true
console.log(isUnicodePalindrome(" ")); // Output: true

```

### Explanation:

1. **Objective:** Validate Unicode strings as palindromes while considering only alphanumeric characters.
2. **Approach:**
  - o Use two pointers to traverse the string.
  - o Skip non-alphanumeric characters, including Unicode symbols, using the isAlphanumericUnicode function.
  - o Compare characters case-insensitively.
3. **Steps:**
  - o Initialize two pointers at the start and end of the string.
  - o Skip invalid characters and compare valid ones.
  - o Return false if characters don't match; otherwise, return true.
4. **Why it Works:** Regular expressions with Unicode support ensure that all alphanumeric characters are correctly identified and processed.
5. **Time Complexity:** O(n) — Single traversal of the string.
6. **Space Complexity:** O(1) — No additional space is used.

### Valid Palindrome (Recursive Approach)

**165. Problem: Validate if a string is a palindrome using recursion, considering only alphanumeric characters and ignoring case.**

#### Example:

Input: s = "A man, a plan, a canal: Panama";  
 Output: true;

Input: s = "race a car";  
 Output: false;

#### Solution:

```

function isPalindromeRecursive(s) {
    function helper(left, right) {
        if (left >= right) return true;

        // Skip non-alphanumeric characters
        if (!isAlphanumeric(s[left])) return helper(left + 1, right);
        if (!isAlphanumeric(s[right])) return helper(left, right - 1);

        // Compare characters case-insensitively
        if (s[left].toLowerCase() !== s[right].toLowerCase()) return false;

        return helper(left + 1, right - 1);
    }

    function isAlphanumeric(char) {
        return /^[a-zA-Z0-9]$/.test(char);
    }

    return helper(0, s.length - 1);
}

// Test
console.log(isPalindromeRecursive("A man, a plan, a canal: Panama")); // Output: true
console.log(isPalindromeRecursive("race a car")); // Output: false
console.log(isPalindromeRecursive(" ")); // Output: true
console.log(isPalindromeRecursive("0P")); // Output: false

```

### **Explanation:**

1. **Objective:** Use recursion to determine if a string is a valid palindrome.
2. **Approach:**
  - o Define a helper function that takes two pointers (left and right) and compares the characters at these positions.
  - o Skip non-alphanumeric characters by recursively moving the pointers inward.
  - o Compare characters case-insensitively and continue until the pointers meet.
3. **Steps:**
  - o If the left pointer is greater than or equal to the right pointer, return true.
  - o Skip non-alphanumeric characters using the helper function.
  - o Compare characters; if they don't match, return false. Otherwise, recursively call the helper function with the updated pointers.
4. **Why it Works:** The recursion ensures that all valid characters are compared symmetrically.
5. **Time Complexity:**  $O(n)$  — Each character is processed once.
6. **Space Complexity:**  $O(n)$  — Space used by the recursion stack.

### **Valid Palindrome (With Maximum Allowed Modifications)**

**166. Problem: Validate if a string can become a palindrome by modifying at most one character.**

### Example:

```
Input: s = "abca"
Output: true
Explanation: You could modify 'c' to 'b' to make it a palindrome.
```

```
Input: s = "abc"
Output: false
```

### Solution:

```
function validPalindromeWithModification(s) {
    function isPalindromeRange(left, right) {
        while (left < right) {
            if (s[left] !== s[right]) return false;
            left++;
            right--;
        }
        return true;
    }

    let left = 0;
    let right = s.length - 1;

    while (left < right) {
        if (s[left] !== s[right]) {
            // Try skipping one character from either side
            return (
                isPalindromeRange(left + 1, right) || isPalindromeRange(left, right - 1)
            );
        }
        left++;
        right--;
    }

    return true;
}

// Test
console.log(validPalindromeWithModification("abca")); // Output: true
console.log(validPalindromeWithModification("abc")); // Output: false
console.log(validPalindromeWithModification("deeee")); // Output: true
console.log(validPalindromeWithModification(""));
```

### Explanation:

1. **Objective:** Check if the string can become a palindrome with at most one character modification.
2. **Approach:**
  - o Use two pointers to traverse the string.

- If characters mismatch, check if skipping one character from either side results in a palindrome.
  - Use a helper function (isPalindromeRange) to validate a substring.
3. **Steps:**
- Traverse the string using two pointers.
  - When a mismatch is found, validate both possible substrings created by skipping one character.
  - Return true if either substring is valid; otherwise, return false.
4. **Why it Works:** Skipping one character at a mismatch ensures that we only modify one character to validate the palindrome.
5. **Time Complexity:** O(n) — Traversal and substring validation take linear time.
6. **Space Complexity:** O(1) — No additional space is used apart from the pointers.

### Valid Palindrome (Palindrome Permutation)

#### 167. Problem: Check if any permutation of a string can form a palindrome.

##### Example:

Input: s = "tactcoa"

Output: true

Explanation: Permutations like "tacocat" or "atcocta" can form a palindrome.

Input: s = "abc"

Output: false

##### Solution:

```
function canPermutePalindrome(s) {
  const charCount = new Map();

  for (const char of s) {
    charCount.set(char, (charCount.get(char) || 0) + 1);
  }

  let oddCount = 0;

  for (const count of charCount.values()) {
    if (count % 2 !== 0) {
      oddCount++;
      if (oddCount > 1) return false;
    }
  }

  return true;
}

// Test
console.log(canPermutePalindrome("tactcoa")); // Output: true
```

```
console.log(canPermutePalindrome("abc")); // Output: false
console.log(canPermutePalindrome("aabb")); // Output: true
console.log(canPermutePalindrome("")); // Output: true
```

### Explanation:

1. **Objective:** Determine if any permutation of the string can be a palindrome.
2. **Approach:**
  - o Count the occurrences of each character in the string.
  - o A string can form a palindrome if at most one character has an odd count.
3. **Steps:**
  - o Traverse the string to count character frequencies using a hash map.
  - o Iterate through the frequency map to check the number of characters with odd counts.
  - o Return true if there is at most one odd count; otherwise, return false.
4. **Why it Works:** A palindrome requires all characters to appear an even number of times, except for at most one character in the middle.
5. **Time Complexity:**  $O(n)$  — Single traversal for counting and checking frequencies.
6. **Space Complexity:**  $O(k)$  — Space for the frequency map, where  $k$  is the number of unique characters.

### Valid Palindrome (Ignore Spaces)

**168. Problem:** **Check if a string is a valid palindrome by ignoring spaces, while considering only alphanumeric characters and ignoring cases.**

### Example:

```
Input: s = "nurses run";
Output: true;
```

```
Input: s = "hello world";
Output: false;
```

### Solution:

```
function isPalindromeSpecialChars(s) {
    let left = 0;
    let right = s.length - 1;

    while (left < right) {
        // Skip non-alphanumeric characters
        while (left < right && !isAlphanumeric(s[left])) {
            left++;
        }
        while (left < right && !isAlphanumeric(s[right])) {
            right--;
        }
        if (s[left].toLowerCase() !== s[right].toLowerCase()) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

```

// Compare characters
if (s[left].toLowerCase() !== s[right].toLowerCase()) {
    return false;
}
left++;
right--;
}

return true;
}

function isAlphanumeric(char) {
    return /^[a-zA-Z0-9]$/.test(char);
}

// Test
console.log(isPalindromeSpecialChars("a@b@a")); // Output: true
console.log(isPalindromeSpecialChars("a!b!c")); // Output: false
console.log(isPalindromeSpecialChars("a!b#A")); // Output: true
console.log(isPalindromeSpecialChars(" ")); // Output: true

```

### **Explanation:**

1. **Objective:** Validate if a string is a palindrome by ignoring spaces and non-alphanumeric characters.
2. **Approach:**
  - o Use two pointers to traverse the string from both ends.
  - o Skip spaces and non-alphanumeric characters.
  - o Compare valid characters case-insensitively.
3. **Steps:**
  - o Initialize two pointers at the start and end of the string.
  - o Skip invalid characters, including spaces, using helper logic.
  - o Compare the remaining characters and return false on any mismatch.
4. **Why it Works:** Skipping irrelevant characters ensures that only the meaningful content is considered.
5. **Time Complexity:**  $O(n)$  — Single traversal of the string.
6. **Space Complexity:**  $O(1)$  — No additional memory used.

### **Valid Palindrome (Handle Strings with Special Characters)**

**169. Problem: Check if a string is a valid palindrome while handling strings with special characters, considering only alphanumeric ones.**

### **Example:**

Input: s = "a@b@a";  
Output: true;

```
Input: s = "a!b!c";
Output: false;
```

### Solution:

```
function isPalindromeSpecialChars(s) {
    let left = 0;
    let right = s.length - 1;

    while (left < right) {
        // Skip non-alphanumeric characters
        while (left < right && !isAlphanumeric(s[left])) {
            left++;
        }
        while (left < right && !isAlphanumeric(s[right])) {
            right--;
        }

        // Compare characters
        if (s[left].toLowerCase() !== s[right].toLowerCase()) {
            return false;
        }
        left++;
        right--;
    }

    return true;
}

function isAlphanumeric(char) {
    return /^[a-zA-Z0-9]$/.test(char);
}

// Test
console.log(isPalindromeSpecialChars("a@b@a")); // Output: true
console.log(isPalindromeSpecialChars("a!b!c")); // Output: false
console.log(isPalindromeSpecialChars("a!b#A")); // Output: true
console.log(isPalindromeSpecialChars(" ")); // Output: true
```

### Explanation:

1. **Objective:** Ensure the function handles special characters by ignoring them during palindrome validation.
2. **Approach:**
  - o Use two pointers to traverse the string from both ends.
  - o Skip special characters and validate only alphanumeric characters.
3. **Steps:**
  - o Traverse the string, ignoring invalid characters.

- Compare valid characters case-insensitively.
  - Return false on any mismatch.
- Why it Works:** Ignoring irrelevant characters ensures accurate palindrome validation.
  - Time Complexity:** O(n) — Single traversal of the string.
  - Space Complexity:** O(1) — No additional space used.

### **Valid Palindrome (Check Longest Palindromic Substring)**

**170. Problem: Find the longest palindromic substring in a given string.**

**Example:**

Input: s = "babad"  
Output: "bab" or "aba"

Input: s = "cbbd"  
Output: "bb"

**Solution:**

```
function longestPalindromeSubstr(s) {
    let start = 0;
    let maxLength = 0;

    function expandAroundCenter(left, right) {
        while (left >= 0 && right < s.length && s[left] === s[right]) {
            left--;
            right++;
        }
        return right - left - 1;
    }

    for (let i = 0; i < s.length; i++) {
        const len1 = expandAroundCenter(i, i); // Odd-length palindromes
        const len2 = expandAroundCenter(i, i + 1); // Even-length palindromes
        const len = Math.max(len1, len2);

        if (len > maxLength) {
            maxLength = len;
            start = i - Math.floor((len - 1) / 2);
        }
    }

    return s.substring(start, start + maxLength);
}

// Test
console.log(longestPalindromeSubstr("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeSubstr("cbbd")); // Output: "bb"
console.log(longestPalindromeSubstr("a")); // Output: "a"
```

```
console.log(longestPalindromeSubstring("ac")); // Output: "a" or "c"
```

### Explanation:

1. **Objective:** Identify the longest palindromic substring in the input string.
2. **Approach:**
  - o Use the expand-around-center technique to check for palindromes from each character as the center.
  - o Check both odd-length and even-length palindromes.
3. **Steps:**
  - o Traverse the string.
  - o For each character, expand outward and calculate the palindrome length.
  - o Track the starting position and maximum length of the longest palindrome.
  - o Extract and return the substring.
4. **Why it Works:** Expanding around centers efficiently captures all possible palindromes.
5. **Time Complexity:**  $O(n^2)$  — Each character is used as a center, with linear expansion per character.
6. **Space Complexity:**  $O(1)$  — No additional space is used.

## Palindromic Substrings (Count All Palindromic Substrings)

**171. Problem:** Given a string s, return the total number of palindromic substrings in it.  
A substring is a contiguous sequence of characters within a string. A substring is palindromic if it reads the same backward as forward.

### Example:

```
Input: s = "abc"
Output: 3
Explanation: Three palindromic substrings are "a", "b", "c".
```

```
Input: s = "aaa"
Output: 6
Explanation: Six palindromic substrings are "a", "a", "a", "aa", "aa", "aaa".
```

### Solution:

```
function countSubstrings(s) {
    let count = 0;

    function expandAroundCenter(left, right) {
        while (left >= 0 && right < s.length && s[left] === s[right]) {
            count++;
            left--;
            right++;
        }
    }

    for (let i = 0; i < s.length; i++) {
        expandAroundCenter(i, i); // Odd length
        expandAroundCenter(i, i + 1); // Even length
    }
}
```

```

        right++;
    }

for (let i = 0; i < s.length; i++) {
    // Odd-length palindromes
    expandAroundCenter(i, i);
    // Even-length palindromes
    expandAroundCenter(i, i + 1);
}

return count;
}

// Test
console.log(countSubstrings("abc")); // Output: 3
console.log(countSubstrings("aaa")); // Output: 6
console.log(countSubstrings("aab")); // Output: 4
console.log(countSubstrings("abcd")); // Output: 4

```

### Explanation:

1. **Objective:** Count all distinct palindromic substrings in the input string.
2. **Approach:**
  - o Use the **expand-around-center** technique to identify palindromes by treating each character as a center.
  - o For each center, expand outward as long as the characters match.
  - o Check for both odd-length (one center) and even-length (two centers) palindromes.
3. **Steps:**
  - o Iterate through the string, treating each character as the center of a palindrome.
  - o Expand outward and count substrings for both odd-length and even-length cases.
  - o Return the total count.
4. **Why it Works:** Each expansion captures all possible palindromic substrings, ensuring no duplicates are missed.
5. **Time Complexity:**  $O(n^2)$  — Expanding around each center takes linear time, and there are  $n$  centers.
6. **Space Complexity:**  $O(1)$  — No additional space is used apart from variables.

### Palindromic Substrings (Using Dynamic Programming)

#### 172. Problem: Count all palindromic substrings in a string using a dynamic programming approach.

##### Example:

Input:  $s = "abc"$ ;  
Output: 3;

Input: s = "aaa";  
Output: 6;

### Solution:

```
function countSubstringsDP(s) {  
    const n = s.length;  
    let count = 0;  
  
    // Create a DP table to store palindrome information  
    const dp = Array.from({ length: n }, () => Array(n).fill(false));  
  
    // Single character substrings are always palindromes  
    for (let i = 0; i < n; i++) {  
        dp[i][i] = true;  
        count++;  
    }  
  
    // Check substrings of length 2  
    for (let i = 0; i < n - 1; i++) {  
        if (s[i] === s[i + 1]) {  
            dp[i][i + 1] = true;  
            count++;  
        }  
    }  
  
    // Check substrings of length greater than 2  
    for (let len = 3; len <= n; len++) {  
        for (let i = 0; i <= n - len; i++) {  
            const j = i + len - 1;  
            if (s[i] === s[j] && dp[i + 1][j - 1]) {  
                dp[i][j] = true;  
                count++;  
            }  
        }  
    }  
  
    return count;  
}  
  
// Test  
console.log(countSubstringsDP("abc")); // Output: 3  
console.log(countSubstringsDP("aaa")); // Output: 6  
console.log(countSubstringsDP("aab")); // Output: 4  
console.log(countSubstringsDP("abcd")); // Output: 4
```

### Explanation:

1. **Objective:** Use a dynamic programming table to count palindromic substrings efficiently.
2. **Approach:**
  - o Use a 2D DP table  $dp[i][j]$  where true means the substring  $s[i:j+1]$  is a palindrome.
  - o Single characters are always palindromes.
  - o Two-character substrings are palindromes if both characters are the same.
  - o For substrings longer than two, a substring is a palindrome if the first and last characters match, and the substring between them is a palindrome.
3. **Steps:**
  - o Fill the DP table for substrings of increasing lengths.
  - o Count valid palindromic substrings as you update the table.
4. **Why it Works:** The DP table avoids redundant calculations by reusing results for smaller substrings.
5. **Time Complexity:**  $O(n^2)$  — Double loop for filling the DP table.
6. **Space Complexity:**  $O(n^2)$  — Space for the DP table.

### Palindromic Substrings (Count Palindromes in Real-Time)

**173. Problem: Modify the expand-around-center solution to print each palindrome as it is counted.**

**Example:**

```
Input: s = "abc";
Output: 3;
Palindromes: ["a", "b", "c"];
```

```
Input: s = "aaa";
Output: 6;
Palindromes: ["a", "a", "a", "aa", "aa", "aaa"];
```

**Solution:**

```
function countAndPrintSubstrings(s) {
  let count = 0;

  function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      console.log(s.substring(left, right + 1)); // Print the palindrome
      count++;
      left--;
      right++;
    }
  }

  for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd-length palindromes
    expandAroundCenter(i, i + 1); // Even-length palindromes
  }
}
```

```

    }

    return count;
}

// Test
console.log(countAndPrintSubstrings("abc")); // Output: 3, Palindromes: ["a", "b", "c"]
console.log(countAndPrintSubstrings("aaa")); // Output: 6, Palindromes: ["a", "a", "a", "aa",
"aa", "aaa"]

```

### Explanation:

1. **Objective:** Extend the expand-around-center solution to display all identified palindromic substrings.
2. **Approach:**
  - o Print each substring as soon as it is identified as a palindrome.
3. **Steps:**
  - o Expand outward from each center and print each substring.
  - o Count the substrings as they are printed.
4. **Why it Works:** Real-time identification ensures clarity and transparency of the solution.
5. **Time Complexity:**  $O(n^2)$  — Expansion takes linear time for each center.
6. **Space Complexity:**  $O(1)$  — No additional memory is used apart from variables.

### Palindromic Substrings (Find Longest Palindromic Substring)

#### 174. Problem: **Find the longest palindromic substring from a given string.**

##### Example:

Input: s = "babad"  
Output: "bab" or "aba"

Input: s = "cbbd"  
Output: "bb"

##### Solution:

```

function longestPalindromeSubstring(s) {
    let start = 0;
    let maxLength = 0;

    function expandAroundCenter(left, right) {
        while (left >= 0 && right < s.length && s[left] === s[right]) {
            left--;
            right++;
        }
        return right - left - 1;
    }
}

```

```

for (let i = 0; i < s.length; i++) {
    const len1 = expandAroundCenter(i, i); // Odd-length palindromes
    const len2 = expandAroundCenter(i, i + 1); // Even-length palindromes
    const len = Math.max(len1, len2);

    if (len > maxLength) {
        maxLength = len;
        start = i - Math.floor((len - 1) / 2);
    }
}

return s.substring(start, start + maxLength);
}

// Test
console.log(longestPalindromeSubstring("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeSubstring("cbbd")); // Output: "bb"
console.log(longestPalindromeSubstring("a")); // Output: "a"
console.log(longestPalindromeSubstring("ac")); // Output: "a" or "c"

```

### **Explanation:**

1. **Objective:** Find the longest substring of the input string that reads the same backward as forward.
2. **Approach:**
  - o Use the **expand-around-center** method, treating each character as a potential center for palindromes.
  - o Expand outward from the center while the substring is valid.
  - o Track the longest palindrome found during the expansion process.
3. **Steps:**
  - o For each character in the string:
    - Check odd-length palindromes by expanding around the single character.
    - Check even-length palindromes by expanding around pairs of characters.
  - o Update the start index and maxLength if a longer palindrome is found.
  - o Extract the longest palindromic substring from the string using start and maxLength.
4. **Why it Works:** The expand-around-center approach ensures all palindromes are evaluated efficiently without generating substrings explicitly.
5. **Time Complexity:**  $O(n^2)$  — Each character is used as a center for expansion.
6. **Space Complexity:**  $O(1)$  — Only variables are used to store indices and lengths.

### **Palindromic Substrings (Count Palindromic Substrings with Special Characters)**

## 175. Problem: Handle palindromic substrings in strings with special characters like spaces and punctuation marks.

### Example:

```
Input: s = "a!b@b!a";
Output: 4;
Palindromes: ["a", "b", "b", "a!b@b!a"];

Input: s = "a b c";
Output: 3;
Palindromes: ["a", "b", "c"];
```

### Solution:

```
function countSubstringsWithSpecialChars(s) {
  function countSubstringsWithSpecialChars(s) {
    const isAlphanumeric = (char) => /^[a-zA-Z0-9]$/.test(char);
    let count = 0;

    function expandAroundCenter(left, right) {
      while (left >= 0 && right < s.length && s[left] === s[right]) {
        if (isAlphanumeric(s[left])) {
          count++;
        }
        left--;
        right++;
      }
    }

    for (let i = 0; i < s.length; i++) {
      expandAroundCenter(i, i); // Odd-length palindromes
      expandAroundCenter(i, i + 1); // Even-length palindromes
    }
  }

  return count;
}

// Test
console.log(countSubstringsWithSpecialChars("a!b@b!a")); // Output: 4
console.log(countSubstringsWithSpecialChars("a b c")); // Output: 3
console.log(countSubstringsWithSpecialChars("!@#$%")); // Output: 0
console.log(countSubstringsWithSpecialChars("abc!cba")); // Output: 7
```

### Explanation:

1. **Objective:** Count all palindromic substrings while considering special characters, ensuring that only meaningful palindromic substrings are counted.
2. **Approach:**
  - o Use the **expand-around-center** method to evaluate potential palindromes.
  - o A helper function `isAlphanumeric` checks whether a character is alphanumeric.
  - o While expanding, include substrings that are valid palindromes and consider only alphanumeric characters when counting.
3. **Steps:**
  - o Loop through each character in the string, treating it as the center of a potential palindrome.
  - o Expand outward for both odd-length and even-length palindromes.
  - o At each valid expansion, check if the characters at the left and right pointers are alphanumeric before counting the substring.
  - o Return the final count.
4. **Edge Cases:**
  - o Strings with no alphanumeric characters (e.g., !@#\$%) should return 0.
  - o Strings with only one character or completely symmetrical substrings (e.g., "a!b@b!a") should count valid palindromes correctly.
5. **Why it Works:** The expansion approach ensures all substrings are checked for palindromic properties while filtering out non-alphanumeric characters for meaningful results.
6. **Time Complexity:**  $O(n^2)$  — Expansion is linear for each center, and there are  $n$  centers.
7. **Space Complexity:**  $O(1)$  — No additional space is used apart from the counter and helper function.

### Palindromic Substrings (Count for All Substrings)

**176. Problem: Count all palindromic substrings by explicitly generating all substrings and checking each one.**

**Example:**

```
Input: s = "abc"
Output: 3
Explanation: Palindromic substrings are "a", "b", "c".
```

```
Input: s = "aaa"
Output: 6
Explanation: Palindromic substrings are "a", "a", "a", "aa", "aa", "aaa".
```

**Solution:**

```
function countSubstringsBruteForce(s) {
  let count = 0;

  function isPalindrome(substring) {
    let left = 0;
    let right = substring.length - 1;
```

```

while (left < right) {
    if (substring[left] !== substring[right]) return false;
    left++;
    right--;
}
return true;
}

for (let i = 0; i < s.length; i++) {
    for (let j = i; j < s.length; j++) {
        if (isPalindrome(s.slice(i, j + 1))) {
            count++;
        }
    }
}

return count;
}

// Test
console.log(countSubstringsBruteForce("abc")); // Output: 3
console.log(countSubstringsBruteForce("aaa")); // Output: 6
console.log(countSubstringsBruteForce("aab")); // Output: 4
console.log(countSubstringsBruteForce("abcd")); // Output: 4

```

### **Explanation:**

1. **Objective:** Use a brute force approach to count all palindromic substrings by generating and checking each substring.
2. **Approach:**
  - o Generate all substrings of s using nested loops.
  - o Check if each substring is a palindrome.
3. **Steps:**
  - o Iterate through all possible starting and ending indices of substrings.
  - o For each substring, check if it is a palindrome.
  - o Increment the count for valid palindromes.
4. **Why it Works:** Explicit generation ensures all substrings are evaluated.
5. **Time Complexity:**  $O(n^3)$  — Two loops for generating substrings and one for checking.
6. **Space Complexity:**  $O(1)$  — No additional space apart from variables.

### **Palindromic Substrings (With At Most One Change)**

**177. Problem:** Count the number of palindromic substrings where at most one character can be changed to make it a palindrome.

#### **Example:**

Input: s = "abca"

Output: 7

Explanation: Palindromic substrings are "a", "b", "c", "a", "aba", "aca", and "abca".

Input: s = "abc"

Output: 3

### Solution:

```
function countSubstringsWithOneChange(s) {  
    let count = 0;  
  
    function expandAroundCenter(left, right, changesAllowed) {  
        while (left >= 0 && right < s.length) {  
            if (s[left] !== s[right]) {  
                if (changesAllowed === 0) break;  
                changesAllowed--;  
            }  
            count++;  
            left--;  
            right++;  
        }  
    }  
  
    for (let i = 0; i < s.length; i++) {  
        expandAroundCenter(i, i, 1); // Odd-length palindromes  
        expandAroundCenter(i, i + 1, 1); // Even-length palindromes  
    }  
  
    return count;  
}  
  
// Test  
console.log(countSubstringsWithOneChange("abca")); // Output: 7  
console.log(countSubstringsWithOneChange("abc")); // Output: 3  
console.log(countSubstringsWithOneChange("aaa")); // Output: 6  
console.log(countSubstringsWithOneChange("abcd")); // Output: 4
```

### Explanation:

1. **Objective:** Count palindromic substrings, allowing at most one mismatch between characters.
2. **Approach:**
  - o Use the expand-around-center technique with an additional parameter to track allowed mismatches.
  - o For each center, count substrings until a mismatch limit is reached.
3. **Steps:**
  - o Expand outward from each center and count valid palindromes.
  - o Stop expansion if the mismatch limit is exceeded.

4. **Why it Works:** The mismatch allowance extends the logic of regular palindromes to account for slight variations.
5. **Time Complexity:**  $O(n^2)$  — Expansion for each center.
6. **Space Complexity:**  $O(1)$  — No additional space is required.

### Palindromic Substrings (With Distinct Values)

**178. Problem: Count the number of distinct palindromic substrings.**

**Example:**

```
Input: s = "aaa"
Output: 3
Explanation: The distinct palindromic substrings are "a", "aa", and "aaa".

Input: s = "ababa"
Output: 6
Explanation: The distinct palindromic substrings are "a", "b", "aba", "bab", "ababa", and "aa".
```

**Solution:**

```
function countDistinctPalindromicSubstrings(s) {
  const palindromes = new Set();

  function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      palindromes.add(s.substring(left, right + 1));
      left--;
      right++;
    }
  }

  for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd-length palindromes
    expandAroundCenter(i, i + 1); // Even-length palindromes
  }

  return palindromes.size;
}

// Test
console.log(countDistinctPalindromicSubstrings("aaa")); // Output: 3
console.log(countDistinctPalindromicSubstrings("ababa")); // Output: 6
console.log(countDistinctPalindromicSubstrings("abc")); // Output: 3
console.log(countDistinctPalindromicSubstrings("abcd")); // Output: 4
```

**Explanation:**

1. **Objective:** Identify all unique palindromic substrings in the string.
2. **Approach:**
  - o Use a Set to store palindromes as they are found.
  - o Expand outward from each center and add substrings to the set.
3. **Steps:**
  - o For each character, expand outward for both odd and even lengths.
  - o Add each identified palindrome to the set.
  - o Return the size of the set.
4. **Why it Works:** The Set ensures duplicates are eliminated.
5. **Time Complexity:**  $O(n^2)$  — Expansion for each center.
6. **Space Complexity:**  $O(n^2)$  — Space for storing substrings in the set.

### **Palindromic Substrings (Using Hash Map for Frequency Count)**

**179. Problem: Count the frequency of distinct palindromic substrings in a string.**

**Example:**

```
Input: s = "aaa"
Output: { "a": 3, "aa": 2, "aaa": 1 }
Explanation: The palindromic substrings are "a" (3 times), "aa" (2 times), and "aaa" (1 time).
```

```
Input: s = "ababa"
Output: { "a": 3, "b": 2, "aba": 2, "bab": 1, "ababa": 1 }
```

**Solution:**

```
function countDistinctPalindromeFrequencies(s) {
  const palindromes = new Map();

  function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      const substring = s.substring(left, right + 1);
      palindromes.set(substring, (palindromes.get(substring) || 0) + 1);
      left--;
      right++;
    }
  }

  for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd-length palindromes
    expandAroundCenter(i, i + 1); // Even-length palindromes
  }

  return Object.fromEntries(palindromes);
}

// Test
```

```

console.log(countDistinctPalindromeFrequencies("aaa")); // Output: { "a": 3, "aa": 2, "aaa": 1
}
console.log(countDistinctPalindromeFrequencies("ababa")); // Output: { "a": 3, "b": 2, "aba": 2, "bab": 1, "ababa": 1 }
console.log(countDistinctPalindromeFrequencies("abc")); // Output: { "a": 1, "b": 1, "c": 1 }
console.log(countDistinctPalindromeFrequencies("abcd")); // Output: { "a": 1, "b": 1, "c": 1, "d": 1 }

```

### Explanation:

1. **Objective:** Count the frequency of distinct palindromic substrings in the input string.
2. **Approach:**
  - o Use the expand-around-center technique to identify palindromes dynamically.
  - o Use a Map to store each unique palindrome as a key and its frequency as the value.
3. **Steps:**
  - o Traverse the string, treating each character as the center of a potential palindrome.
  - o For each valid palindrome found during expansion, update its count in the Map.
  - o Convert the Map to an object for the final output.
4. **Edge Cases:**
  - o For strings with all identical characters (e.g., "aaa"), ensure all substrings are counted.
  - o For strings with no palindromes longer than one character, count only individual characters.
5. **Why it Works:** The Map ensures each palindrome is tracked uniquely, while the expand-around-center approach finds all palindromes efficiently.
6. **Time Complexity:**  $O(n^2)$  — Each center requires linear expansion.
7. **Space Complexity:**  $O(n^2)$  — Space used for storing palindromic substrings in the Map.

## Palindromic Substrings (Longest Palindromic Substring Using Dynamic Programming)

### 180. Problem: Find the longest palindromic substring in a string using a dynamic programming approach.

#### Example:

Input: s = "babad"  
Output: "bab" or "aba"

Input: s = "cbbd"  
Output: "bb"

#### Solution:

```
function longestPalindromeDP(s) {
```

```

const n = s.length;
if (n === 0) return "";

let start = 0;
let maxLength = 1;

// Create a DP table
const dp = Array.from({ length: n }, () => Array(n).fill(false));

// All single characters are palindromes
for (let i = 0; i < n; i++) {
  dp[i][i] = true;
}

// Check substrings of length 2
for (let i = 0; i < n - 1; i++) {
  if (s[i] === s[i + 1]) {
    dp[i][i + 1] = true;
    start = i;
    maxLength = 2;
  }
}

// Check substrings of length > 2
for (let len = 3; len <= n; len++) {
  for (let i = 0; i <= n - len; i++) {
    const j = i + len - 1;
    if (s[i] === s[j] && dp[i + 1][j - 1]) {
      dp[i][j] = true;
      start = i;
      maxLength = len;
    }
  }
}

return s.substring(start, start + maxLength);
}

// Test
console.log(longestPalindromeDP("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeDP("cbbd")); // Output: "bb"
console.log(longestPalindromeDP("a")); // Output: "a"
console.log(longestPalindromeDP("ac")); // Output: "a" or "c"

```

### Explanation:

1. **Objective:** Use dynamic programming to efficiently find the longest palindromic substring.
2. **Approach:**

- Create a DP table  $dp[i][j]$  where true means the substring  $s[i:j+1]$  is a palindrome.
- Single characters are palindromes by default.
- Substrings of length 2 are palindromes if both characters are the same.
- For substrings of length  $> 2$ , a substring is a palindrome if the first and last characters are the same, and the substring between them is also a palindrome.

### 3. Steps:

- Initialize the DP table for single characters and two-character substrings.
- Fill the table for substrings of increasing lengths.
- Track the starting index and length of the longest palindrome.
- Extract the substring using the starting index and length.

### 4. Edge Cases:

- For empty strings, return an empty substring.
- For strings with all identical characters, return the entire string.

5. Why it Works: The DP table stores intermediate results, avoiding redundant computations for overlapping substrings.

6. Time Complexity:  $O(n^2)$  — Double loop for filling the DP table.

7. Space Complexity:  $O(n^2)$  — Space for the DP table.

## Longest Palindromic Substring (Expand Around Center)

**181. Problem: Find the longest palindromic substring in a given string using the "Expand Around Center" approach.**

### Example:

Input:  $s = "babad"$   
Output: "bab" or "aba"

Input:  $s = "cbbd"$   
Output: "bb"

### Solution:

```
function longestPalindromeExpandCenter(s) {
    if (!s || s.length < 1) return "";

    let start = 0;
    let end = 0;

    function expandAroundCenter(left, right) {
        while (left >= 0 && right < s.length && s[left] === s[right]) {
            left--;
            right++;
        }
        return right - left - 1; // Length of the palindrome
    }

    for (let i = 0; i < s.length; i++) {
```

```

const len1 = expandAroundCenter(i, i); // Odd-length palindromes
const len2 = expandAroundCenter(i, i + 1); // Even-length palindromes
const maxLen = Math.max(len1, len2);

if (maxLen > end - start) {
    start = i - Math.floor((maxLen - 1) / 2);
    end = i + Math.floor(maxLen / 2);
}

return s.substring(start, end + 1);
}

// Test
console.log(longestPalindromeExpandCenter("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeExpandCenter("cbbd")); // Output: "bb"
console.log(longestPalindromeExpandCenter("a")); // Output: "a"
console.log(longestPalindromeExpandCenter("ac")); // Output: "a" or "c"

```

### **Explanation:**

1. **Objective:** Find the longest substring that is a palindrome by expanding outward from each possible center.
2. **Approach:**
  - o Treat each character and each pair of adjacent characters as potential centers of palindromes.
  - o Expand outward as long as the characters at the left and right pointers match.
  - o Keep track of the longest palindrome found during the process.
3. **Steps:**
  - o Iterate through each character in the string as a potential center.
  - o Expand outward for both odd-length and even-length palindromes.
  - o Update the start and end indices of the longest palindrome when a longer one is found.
  - o Return the substring using the final start and end indices.
4. **Why it Works:** The symmetric property of palindromes ensures that all palindromic substrings can be found by expanding around their centers.
5. **Time Complexity:**  $O(n^2)$  — Each character is used as a center, and expansion is linear for each center.
6. **Space Complexity:**  $O(1)$  — No additional space is used apart from variables.

### **Longest Palindromic Substring (Dynamic Programming Approach)**

#### **182. Problem: Find the longest palindromic substring in a string using dynamic programming.**

##### **Example:**

Input:  $s = \text{"babad"}$   
Output: "bab" or "aba"

Input: s = "cbbd"  
Output: "bb"

### Solution:

```
function longestPalindromeDP(s) {
    const n = s.length;
    if (n === 0) return "";

    let start = 0;
    let maxLength = 1;

    // Create a DP table
    const dp = Array.from({ length: n }, () => Array(n).fill(false));

    // All single characters are palindromes
    for (let i = 0; i < n; i++) {
        dp[i][i] = true;
    }

    // Check substrings of length 2
    for (let i = 0; i < n - 1; i++) {
        if (s[i] === s[i + 1]) {
            dp[i][i + 1] = true;
            start = i;
            maxLength = 2;
        }
    }

    // Check substrings of length greater than 2
    for (let len = 3; len <= n; len++) {
        for (let i = 0; i <= n - len; i++) {
            const j = i + len - 1;
            if (s[i] === s[j] && dp[i + 1][j - 1]) {
                dp[i][j] = true;
                start = i;
                maxLength = len;
            }
        }
    }

    return s.substring(start, start + maxLength);
}

// Test
console.log(longestPalindromeDP("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeDP("cbbd")); // Output: "bb"
console.log(longestPalindromeDP("a")); // Output: "a"
```

```
console.log(longestPalindromeDP("ac")); // Output: "a" or "c"
```

### Explanation:

1. **Objective:** Use a DP table to efficiently find the longest palindromic substring by storing whether substrings are palindromes.
2. **Approach:**
  - o Use a 2D DP table  $dp[i][j]$ , where true indicates that the substring  $s[i:j+1]$  is a palindrome.
  - o Single characters ( $s[i:i]$ ) are palindromes.
  - o Substrings of length 2 ( $s[i:i+1]$ ) are palindromes if both characters are the same.
  - o For longer substrings,  $s[i:j]$  is a palindrome if the first and last characters match, and the substring between them ( $s[i+1:j-1]$ ) is also a palindrome.
3. **Steps:**
  - o Initialize the DP table for single-character and two-character substrings.
  - o Fill the DP table for substrings of increasing lengths.
  - o Track the starting index and length of the longest palindrome during the process.
  - o Extract the substring using the final starting index and length.
4. **Why it Works:** The DP table avoids redundant calculations by reusing results for smaller substrings.
5. **Time Complexity:**  $O(n^2)$  — Double loop for filling the DP table.
6. **Space Complexity:**  $O(n^2)$  — Space for the DP table.

### Longest Palindromic Substring (Brute Force)

**183. Problem:** Find the longest palindromic substring by explicitly generating all substrings and checking if each one is a palindrome.

#### Example:

```
Input: s = "babad"
Output: "bab" or "aba"
```

```
Input: s = "cbbd"
Output: "bb"
```

#### Solution:

```
function longestPalindromeBruteForce(s) {
  const isPalindrome = (sub) => {
    let left = 0,
      right = sub.length - 1;
    while (left < right) {
      if (sub[left] !== sub[right]) return false;
      left++;
      right--;
    }
    return true;
  };
  let maxLen = 1;
  let start = 0;
  for (let i = 0; i < s.length; i++) {
    for (let j = i + 1; j < s.length; j++) {
      if (isPalindrome(s.substring(i, j))) {
        if (j - i + 1 > maxLen) {
          maxLen = j - i + 1;
          start = i;
        }
      }
    }
  }
  return s.substring(start, start + maxLen);
}
```

```

    }
    return true;
};

let maxLength = 0;
let longest = "";

for (let i = 0; i < s.length; i++) {
  for (let j = i; j < s.length; j++) {
    const substring = s.substring(i, j + 1);
    if (isPalindrome(substring) && substring.length > maxLength) {
      maxLength = substring.length;
      longest = substring;
    }
  }
}

return longest;
}

// Test
console.log(longestPalindromeBruteForce("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeBruteForce("cbbd")); // Output: "bb"
console.log(longestPalindromeBruteForce("a")); // Output: "a"
console.log(longestPalindromeBruteForce("ac")); // Output: "a" or "c"

```

### Explanation:

1. **Objective:** Solve the problem by exhaustively checking all substrings for palindromic properties.
2. **Approach:**
  - o Generate all possible substrings using two nested loops.
  - o Use a helper function (`isPalindrome`) to check if each substring is a palindrome.
  - o Track the longest palindrome found during the process.
3. **Steps:**
  - o Iterate over all possible starting and ending indices for substrings.
  - o For each substring, check if it is a palindrome.
  - o Update the result if a longer palindrome is found.
4. **Why it Works:** This method ensures that all possible substrings are considered, guaranteeing correctness.
5. **Time Complexity:**  $O(n^3)$  — Generating substrings and checking each for palindromic properties is cubic in complexity.
6. **Space Complexity:**  $O(1)$  — No additional space is used apart from variables.

### Longest Palindromic Substring (Iterative Two-Pointer Method)

**184. Problem:** Find the longest palindromic substring using an iterative two-pointer approach that focuses on expanding from the edges inward.

**Example:**

Input: s = "babad"  
Output: "bab" or "aba"

Input: s = "cbbd"  
Output: "bb"

**Solution:**

```
function longestPalindromeTwoPointer(s) {  
    let longest = "";  
  
    for (let i = 0; i < s.length; i++) {  
        for (let j = i; j < s.length; j++) {  
            if (isPalindrome(s, i, j) && j - i + 1 > longest.length) {  
                longest = s.substring(i, j + 1);  
            }  
        }  
    }  
  
    function isPalindrome(s, left, right) {  
        while (left < right) {  
            if (s[left] !== s[right]) {  
                return false;  
            }  
            left++;  
            right--;  
        }  
        return true;  
    }  
  
    return longest;  
}  
  
// Test  
console.log(longestPalindromeTwoPointer("babad")); // Output: "bab" or "aba"  
console.log(longestPalindromeTwoPointer("cbbd")); // Output: "bb"  
console.log(longestPalindromeTwoPointer("a")); // Output: "a"  
console.log(longestPalindromeTwoPointer("ac")); // Output: "a" or "c"
```

**Explanation:**

1. **Objective:** Identify the longest palindromic substring by iterating over all possible substrings and validating their symmetry using a two-pointer method.

## 2. Approach:

- Use two nested loops to generate all substrings.
- For each substring, use the two-pointer technique to check if it is a palindrome.
- Track the longest palindrome found during the iterations.

## 3. Steps:

- Use i and j as the starting and ending indices for substrings.
- For each substring, check its symmetry by comparing characters at the left and right pointers.
- Update the longest substring if a new one is found.

4. Why it Works: By explicitly validating each substring, this method ensures accuracy, though it is less efficient than optimized solutions.

5. Time Complexity:  $O(n^3)$  — Two loops for substrings and one for palindrome checking.

6. Space Complexity:  $O(1)$  — No additional space is used apart from variables.

## Longest Palindromic Substring (Using Hash Map for Frequency)

### 185. Problem:

**Find the frequency of the longest palindromic substring in the string.**

#### Example:

```
Input: s = "babad"
Output: { "bab": 1 }

Input: s = "aabcbaa"
Output: { "abcba": 1 }
```

#### Solution:

```
function longestPalindromeFrequency(s) {
  const map = new Map();
  let longest = "";

  function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      const substring = s.substring(left, right + 1);
      if (substring.length > longest.length) {
        longest = substring;
      }
      left--;
      right++;
    }
  }

  for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd-length palindromes
    expandAroundCenter(i, i + 1); // Even-length palindromes
  }
}
```

```

    }

// Add the longest palindrome to the map with frequency 1
map.set(longest, 1);

return Object.fromEntries(map);
}

// Test
console.log(longestPalindromeFrequency("babad")); // Output: { "bab": 1 }
console.log(longestPalindromeFrequency("aabcbaa")); // Output: { "abcba": 1 }
console.log(longestPalindromeFrequency("cbbd")); // Output: { "bb": 1 }
console.log(longestPalindromeFrequency("abcd")); // Output: { "a": 1 }

```

### Explanation:

1. **Objective:** Use the expand-around-center approach to find the longest palindromic substring and store its frequency in a hash map.
2. **Approach:**
  - o Expand outward from each center to identify palindromes.
  - o Track the longest palindrome found during the process.
  - o Store the longest palindrome in a hash map with its frequency.
3. **Steps:**
  - o Use the center-expansion technique for both odd and even-length palindromes.
  - o Update the longest palindrome dynamically as new ones are found.
  - o Store the final result in a hash map with frequency 1.
4. **Why it Works:** By focusing only on the longest palindrome and its frequency, this approach combines efficient identification with storage.
5. **Time Complexity:**  $O(n^2)$  — Each center requires linear expansion.
6. **Space Complexity:**  $O(n)$  — Space for storing the result in a map.

### Longest Palindromic Substring (Recursive Solution)

#### 186. Problem: Find the longest palindromic substring using recursion.

##### Example:

Input: s = "babad"  
Output: "bab" or "aba"

Input: s = "cbbd"  
Output: "bb"

##### Solution:

```

function longestPalindromeRecursive(s) {
  if (s.length <= 1) return s;

  function isPalindrome(substring) {

```

```

let left = 0;
let right = substring.length - 1;
while (left < right) {
  if (substring[left] !== substring[right]) {
    return false;
  }
  left++;
  right--;
}
return true;
}

function findLongest(left, right) {
  if (left > right) return "";
  const current = s.substring(left, right + 1);
  if (isPalindrome(current)) return current;

  const excludeLeft = findLongest(left + 1, right);
  const excludeRight = findLongest(left, right - 1);

  return excludeLeft.length > excludeRight.length
    ? excludeLeft
    : excludeRight;
}

return findLongest(0, s.length - 1);
}

// Test
console.log(longestPalindromeRecursive("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeRecursive("cbbd")); // Output: "bb"
console.log(longestPalindromeRecursive("a")); // Output: "a"
console.log(longestPalindromeRecursive("ac")); // Output: "a" or "c"

```

### Explanation:

1. **Objective:** Use recursion to find the longest palindromic substring by reducing the problem size in each call.
2. **Approach:**
  - o Use a helper function to check if a substring is a palindrome.
  - o Recursively explore two possibilities: exclude the leftmost or rightmost character.
  - o Return the longer of the two results.
3. **Steps:**
  - o Base case: If the substring length is 1 or 0, it is a palindrome.
  - o Recursive case: Check the current substring. If it's a palindrome, return it; otherwise, recurse with reduced bounds.
4. **Why it Works:** Recursion breaks down the problem into smaller subproblems, evaluating all possible substrings.

5. **Time Complexity:**  $O(2^n)$  — Recursively explores all substring possibilities.
6. **Space Complexity:**  $O(n)$  — Space for the recursion stack.

### Longest Palindromic Substring (Iterative DP with Space Optimization)

**187. Problem:** **Find the longest palindromic substring using a dynamic programming approach but optimize space complexity by only using a 1D array.**

**Example:**

Input: s = "babad"  
Output: "bab" or "aba"

Input: s = "cbbd"  
Output: "bb"

**Solution:**

```
function longestPalindromeDPOptimized(s) {
  const n = s.length;
  if (n === 0) return "";

  let start = 0;
  let maxLength = 1;

  // Use a 1D array to store the current state
  const dp = Array(n).fill(false);

  for (let i = n - 1; i >= 0; i--) {
    for (let j = i; j < n; j++) {
      if (s[i] === s[j] && (j - i < 3 || dp[j - 1])) {
        dp[j] = true;

        if (j - i + 1 > maxLength) {
          start = i;
          maxLength = j - i + 1;
        }
      } else {
        dp[j] = false;
      }
    }
  }

  return s.substring(start, start + maxLength);
}

// Test
console.log(longestPalindromeDPOptimized("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeDPOptimized("cbbd")); // Output: "bb"
console.log(longestPalindromeDPOptimized("a")); // Output: "a"
```

```
console.log(longestPalindromeDPOptimized("ac")); // Output: "a" or "c"
```

### Explanation:

1. **Objective:** Use a dynamic programming approach to find the longest palindromic substring while reducing space complexity to O(n).
2. **Approach:**
  - o Use a 1D array dp where dp[j] indicates whether the substring s[i:j+1] is a palindrome.
  - o Iterate backward for the starting index i and forward for the ending index j.
  - o Update dp[j] based on whether s[i] equals s[j] and the substring between them is a palindrome.
3. **Steps:**
  - o Initialize dp to store the palindromic state for each substring ending at j.
  - o Update dp[j] during iterations and track the longest palindrome.
4. **Why it Works:** By iterating backward for i, the previous state for dp[j-1] is correctly reused during each update.
5. **Time Complexity:**  $O(n^2)$  — Double loop for filling the DP array.
6. **Space Complexity:**  $O(n)$  — Space used for the dp array.

### Longest Palindromic Substring (Manacher's Algorithm)

**188. Problem:** **Find the longest palindromic substring using Manacher's algorithm, which improves time complexity to O(n).**

#### Example:

```
Input: s = "babad"
Output: "bab" or "aba"
```

```
Input: s = "cbbd"
Output: "bb"
```

#### Solution:

```
function longestPalindromeManacher(s) {
  const preprocess = (s) => {
    const chars = ["#"];
    for (const char of s) {
      chars.push(char, "#");
    }
    return chars;
  };

  const chars = preprocess(s);
  const n = chars.length;
  const p = Array(n).fill(0);
  let center = 0,
```

```

right = 0;
let maxLen = 0,
start = 0;

for (let i = 0; i < n; i++) {
    const mirror = 2 * center - i;

    if (i < right) {
        p[i] = Math.min(right - i, p[mirror]);
    }

    while (
        i + p[i] + 1 < n &&
        i - p[i] - 1 >= 0 &&
        chars[i + p[i] + 1] === chars[i - p[i] - 1]
    ) {
        p[i]++;
    }

    if (i + p[i] > right) {
        center = i;
        right = i + p[i];
    }

    if (p[i] > maxLen) {
        maxLen = p[i];
        start = (i - p[i]) / 2;
    }
}

return s.substring(start, start + maxLen);
}

// Test
console.log(longestPalindromeManacher("babad")); // Output: "bab" or "aba"
console.log(longestPalindromeManacher("cbbd")); // Output: "bb"
console.log(longestPalindromeManacher("a")); // Output: "a"
console.log(longestPalindromeManacher("ac")); // Output: "a" or "c"

```

### Explanation:

1. **Objective:** Use Manacher's algorithm to efficiently find the longest palindromic substring in linear time.
2. **Approach:**
  - o Preprocess the string by inserting # between each character to handle both odd and even-length palindromes uniformly.
  - o Use a palindrome radius array p to track the length of the palindrome centered at each character.
  - o Use a center and right variable to track the rightmost palindrome boundary.

- For each character, expand around it to find the maximum palindrome radius.
3. **Steps:**
- Preprocess the string to simplify handling of even-length palindromes.
  - Traverse the preprocessed string and calculate the palindrome radius for each character.
  - Track the maximum radius and use it to compute the longest palindromic substring in the original string.
4. **Why it Works:** The preprocessing step ensures that all palindromes can be treated uniformly, and the radius array allows efficient palindrome expansion.
5. **Time Complexity:**  $O(n)$  — Single traversal of the preprocessed string.
6. **Space Complexity:**  $O(n)$  — Space used for the preprocessed string and the radius array.

### Longest Palindromic Substring (Find All Palindromic Substrings)

**189. Problem: Find and return all distinct palindromic substrings in a given string, along with the longest palindromic substring.**

```
Input: s = "babad"
Output: { longest: "bab", allPalindromes: ["b", "a", "bab", "d"] }

Input: s = "cbbd"
Output: { longest: "bb", allPalindromes: ["c", "b", "bb", "d"] }
```

#### Solution:

```
function findAllPalindromicSubstrings(s) {
  const palindromes = new Set();
  let longest = "";

  function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      const substring = s.substring(left, right + 1);
      palindromes.add(substring);
      if (substring.length > longest.length) {
        longest = substring;
      }
      left--;
      right++;
    }
  }

  for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd-length palindromes
    expandAroundCenter(i, i + 1); // Even-length palindromes
  }

  return { longest, allPalindromes: Array.from(palindromes) };
}
```

```
// Test
console.log(findAllPalindromicSubstrings("babad")); // Output: { longest: "bab",
allPalindromes: ["b", "a", "bab", "aba", "d"] }
console.log(findAllPalindromicSubstrings("cbbd")); // Output: { longest: "bb",
allPalindromes: ["c", "b", "bb", "d"] }
console.log(findAllPalindromicSubstrings("aaa")); // Output: { longest: "aaa",
allPalindromes: ["a", "aa", "aaa"] }
console.log(findAllPalindromicSubstrings("a")); // Output: { longest: "a", allPalindromes:
["a"] }
```

### Explanation:

1. **Objective:** Identify all palindromic substrings in the input string and track the longest one.
2. **Approach:**
  - o Use the **expand-around-center** technique to identify palindromes for each character in the string.
  - o Store all palindromes in a Set to eliminate duplicates.
  - o Keep track of the longest palindrome while adding substrings to the set.
3. **Steps:**
  - o Treat each character and adjacent pairs as potential centers.
  - o Expand outward from each center and add valid palindromes to the set.
  - o Update the longest palindrome when a longer one is found.
  - o Return the longest palindrome and all distinct palindromes.
4. **Why it Works:** By expanding around centers and using a Set, this method ensures all valid palindromes are counted without duplication.
5. **Time Complexity:**  $O(n^2)$  — Each center requires linear expansion.
6. **Space Complexity:**  $O(n^2)$  — Space for storing substrings in the set.

### Longest Palindromic Substring (Count Palindromes and Return Longest)

**190. Problem:** Count the total number of palindromic substrings in a given string and return the longest palindromic substring.

#### Example:

Input: s = "babad"  
Output: { count: 7, longest: "bab" }

Input: s = "cbbd"  
Output: { count: 4, longest: "bb" }

#### Solution:

```
function countAndFindLongestPalindrome(s) {
  let count = 0;
  let longest = "";
```

```

function expandAroundCenter(left, right) {
    while (left >= 0 && right < s.length && s[left] === s[right]) {
        const substring = s.substring(left, right + 1);
        count++;
        if (substring.length > longest.length) {
            longest = substring;
        }
        left--;
        right++;
    }
}

for (let i = 0; i < s.length; i++) {
    expandAroundCenter(i, i); // Odd-length palindromes
    expandAroundCenter(i, i + 1); // Even-length palindromes
}

return { count, longest };
}

// Test
console.log(countAndFindLongestPalindrome("babad")); // Output: { count: 7, longest: "bab" }
console.log(countAndFindLongestPalindrome("cbbd")); // Output: { count: 4, longest: "bb" }
console.log(countAndFindLongestPalindrome("aaa")); // Output: { count: 6, longest: "aaa" }
console.log(countAndFindLongestPalindrome("a")); // Output: { count: 1, longest: "a" }

```

### **Explanation:**

1. **Objective:** Count all palindromic substrings in the string while identifying the longest palindrome.
2. **Approach:**
  - o Use the **expand-around-center** technique to identify palindromic substrings.
  - o Increment a counter for every valid palindrome.
  - o Keep track of the longest palindrome dynamically during the process.
3. **Steps:**
  - o Iterate over the string, treating each character as a center for odd-length palindromes.
  - o Also, treat adjacent pairs as centers for even-length palindromes.
  - o Expand outward and count valid palindromes, updating the longest palindrome when necessary.
4. **Why it Works:** By combining counting and tracking of the longest palindrome, this approach achieves both tasks efficiently.
5. **Time Complexity:**  $O(n^2)$  — Each center requires linear expansion.
6. **Space Complexity:**  $O(1)$  — No additional data structures are used apart from variables.

### **Encoding & Decoding Text-Based Data (Basic Encoding and Decoding)**

**191. Problem: Design an encoding and decoding mechanism for a list of strings. The encoded string should be a single string that can be decoded back into the original list of strings.**

**Example:**

```
Input: strings = ["hello", "world"];
Encoded: "5#hello5#world";
Decoded: ["hello", "world"];
```

**Solution:**

```
class Codec {
    encode(strs) {
        return strs.map((str) => `${str.length}#${str}`).join("");
    }

    decode(s) {
        const result = [];
        let i = 0;

        while (i < s.length) {
            let j = i;
            while (s[j] !== "#") {
                j++;
            }

            const length = parseInt(s.slice(i, j));
            const str = s.slice(j + 1, j + 1 + length);
            result.push(str);

            i = j + 1 + length;
        }

        return result;
    }
}

// Test
const codec = new Codec();
const encoded = codec.encode(["hello", "world"]);
console.log("Encoded:", encoded); // Output: "5#hello5#world"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["hello", "world"]
```

**Explanation:**

1. **Objective:** Design a reversible encoding mechanism for a list of strings such that no ambiguity arises during decoding.
2. **Approach:**
  - For **encoding**, prefix each string with its length followed by a delimiter (e.g., #).
  - For **decoding**, parse the string to extract the length and then extract the corresponding substring.
3. **Steps for Encoding:**
  - Iterate through the list of strings.
  - For each string, concatenate its length, a delimiter (#), and the string itself.
  - Join all encoded parts into a single string.
4. **Steps for Decoding:**
  - Traverse the encoded string and extract each substring using the length prefix.
  - Use the delimiter to identify the length and extract the substring of that length.
5. **Why it Works:** Using a length prefix ensures that even strings containing special characters or delimiters can be encoded and decoded without ambiguity.
6. **Time Complexity:**
  - **Encoding:** O(n) — Traverses all strings and their characters.
  - **Decoding:** O(n) — Processes each character in the encoded string.
7. **Space Complexity:** O(n) — Additional space for the encoded/decoded string.

### Encoding & Decoding Text-Based Data (Handle Empty Strings)

**192. Problem:** Extend the encoding and decoding mechanism to handle empty strings in the list.

**Example:**

```
Input: strings = ["hello", "", "world"];
Encoded: "5#hello0#5#world";
Decoded: ["hello", "", "world"];
```

**Solution:**

```
class Codec {
  encode(strs) {
    return strs.map((str) => `${str.length}#${str}`).join("");
  }

  decode(s) {
    const result = [];
    let i = 0;

    while (i < s.length) {
      let j = i;
      while (s[j] !== "#") {
        j++;
      }
      result.push(s.substring(i, j - 1));
      i = j + 1;
    }
    return result;
  }
}
```

```

const length = parseInt(s.slice(i, j));
const str = s.slice(j + 1, j + 1 + length);
result.push(str);

    i = j + 1 + length;
}

return result;
}
}

// Test
const codec = new Codec();
const encoded = codec.encode(["hello", "", "world"]);
console.log("Encoded:", encoded); // Output: "5#hello0#5#world"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["hello", "", "world"]

```

### **Explanation:**

1. **Objective:** Ensure the encoding and decoding mechanism handles empty strings without ambiguity.
2. **Approach:**
  - o During **encoding**, include the length prefix (0#) even for empty strings.
  - o During **decoding**, correctly interpret the 0# prefix to append an empty string to the result.
3. **Steps for Encoding:**
  - o Iterate through the list of strings.
  - o For each string, concatenate its length (0 for empty strings), a delimiter (#), and the string itself.
4. **Steps for Decoding:**
  - o Traverse the encoded string using the same logic as before.
  - o A prefix of 0# corresponds to an empty string, which is appended to the result.
5. **Why it Works:** By consistently using the length prefix, empty strings are uniquely identified during decoding.
6. **Time Complexity:**
  - o **Encoding:** O(n) — Each character in the input strings is processed.
  - o **Decoding:** O(n) — Each character in the encoded string is processed.
7. **Space Complexity:** O(n) — Additional space for the encoded/decoded string.

### **Encoding & Decoding Text-Based Data (Support Delimiters in Strings)**

**193. Problem:** Handle strings containing the delimiter (#) without breaking the encoding and decoding process.

#### **Example:**

Input: strings = ["he#llo", "world"];

```
Encoded: "6#he#llo5#world";
Decoded: ["he#llo", "world"];
```

### Solution:

```
class Codec {
  encode(strs) {
    return strs.map((str) => `${str.length}#${str}`).join("");
  }

  decode(s) {
    const result = [];
    let i = 0;

    while (i < s.length) {
      let j = i;
      while (s[j] !== "#") {
        j++;
      }

      const length = parseInt(s.slice(i, j));
      const str = s.slice(j + 1, j + 1 + length);
      result.push(str);

      i = j + 1 + length;
    }

    return result;
  }
}

// Test
const codec = new Codec();
const encoded = codec.encode(["he#llo", "world"]);
console.log("Encoded:", encoded); // Output: "6#he#llo5#world"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["he#llo", "world"]
```

### Explanation:

1. **Objective:** Support strings containing the delimiter (#) by relying on the length prefix rather than the delimiter itself for decoding.
2. **Approach:**
  - o During **encoding**, include the string length before the delimiter.
  - o During **decoding**, use the length prefix to determine the substring length and ignore any # characters within the substring.
3. **Steps for Encoding:**

- Encode each string with its length prefix followed by the string itself.
  - Join all encoded segments into a single string.
4. **Steps for Decoding:**
    - Parse the length prefix to determine the number of characters in the substring.
    - Extract the substring based on its length, regardless of its content.
  5. **Why it Works:** The length prefix removes reliance on # as a unique delimiter, allowing it to appear naturally in the strings.
  6. **Time Complexity:**
    - **Encoding:** O(n) — Processes each character in the input strings.
    - **Decoding:** O(n) — Processes each character in the encoded string.
  7. **Space Complexity:** O(n) — Space for the encoded/decoded string.

### **Encoding & Decoding Text-Based Data (Support for Unicode Characters)**

**194. Problem: Extend the encoding and decoding mechanism to handle strings containing Unicode characters, ensuring that no special characters or multi-byte sequences cause decoding issues.**

#### **Example:**

```
Input: strings = ["你好", "🌍"];
Encoded: "2#你好1#🌍";
Decoded: ["你好", "🌍"];
```

#### **Solution:**

```
class Codec {
  encode(strs) {
    return strs.map((str) => `${str.length}#${str}`).join("");
  }

  decode(s) {
    const result = [];
    let i = 0;

    while (i < s.length) {
      let j = i;
      while (s[j] !== "#") {
        j++;
      }

      const length = parseInt(s.slice(i, j));
      const str = s.slice(j + 1, j + 1 + length);
      result.push(str);

      i = j + 1 + length;
    }
  }
}
```

```

        return result;
    }

}

// Test
const codec = new Codec();
const encoded = codec.encode(["你好", "🌍"]);
console.log("Encoded:", encoded); // Output: "2#你好1#🌍"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["你好", "🌍"]

```

### Explanation:

1. **Objective:** Ensure that the encoding and decoding mechanism works seamlessly with Unicode strings, including emojis and characters from non-Latin alphabets.
2. **Approach:**
  - o During **encoding**, determine the correct length of each string using JavaScript's native length property, which supports Unicode characters.
  - o During **decoding**, parse the length prefix and extract the substring accordingly, relying on the length to handle multi-byte sequences correctly.
3. **Steps for Encoding:**
  - o For each string, compute its length and prefix it with the delimiter (#) before concatenating it to the result.
4. **Steps for Decoding:**
  - o Parse the length prefix to determine how many characters to extract.
  - o Use substring to extract the Unicode-aware portion of the string.
5. **Why it Works:** JavaScript's string operations are Unicode-compliant, ensuring that the length and slicing methods handle multi-byte sequences without issues.
6. **Time Complexity:**
  - o **Encoding:** O(n) — Processes each character in the input strings.
  - o **Decoding:** O(n) — Processes each character in the encoded string.
7. **Space Complexity:** O(n) — Space for the encoded/decoded string.

### Encoding & Decoding Text-Based Data (Nested Strings)

**195. Problem:** Handle cases where the strings themselves may include encoded strings. The mechanism should be robust enough to encode and decode nested structures.

### Example:

```

Input: strings = ["5#hello", "world"];
Encoded: "7#5#hello5#world";
Decoded: ["5#hello", "world"];

```

### Solution:

```

class Codec {
  encode(strs) {
    return strs.map((str) => `${str.length}#${str}`).join("");
  }

  decode(s) {
    const result = [];
    let i = 0;

    while (i < s.length) {
      let j = i;
      while (s[j] !== "#") {
        j++;
      }

      const length = parseInt(s.slice(i, j));
      const str = s.slice(j + 1, j + 1 + length);
      result.push(str);

      i = j + 1 + length;
    }

    return result;
  }
}

// Test
const codec = new Codec();
const encoded = codec.encode(["5#hello", "world"]);
console.log("Encoded:", encoded); // Output: "7#5#hello5#world"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["5#hello", "world"]

```

### Explanation:

1. **Objective:** Ensure the encoding mechanism is robust enough to handle strings that include already-encoded substrings or delimiters.
2. **Approach:**
  - o Use a length prefix to uniquely identify the boundaries of each string, regardless of its content.
  - o Ignore the contents of the string during decoding and rely solely on the length prefix.
3. **Steps for Encoding:**
  - o Compute the length of each string and concatenate it with the delimiter (#) and the string itself.
4. **Steps for Decoding:**
  - o Parse the length prefix to determine the exact substring to extract, ignoring any delimiters within the string.

5. **Why it Works:** The length prefix provides unambiguous boundaries for each string, even if they include encoded content or delimiters.
6. **Time Complexity:**
  - o **Encoding:** O(n) — Processes each character in the input strings.
  - o **Decoding:** O(n) — Processes each character in the encoded string.
7. **Space Complexity:** O(n) — Space for the encoded/decoded string.

### Encoding & Decoding Text-Based Data (Error Handling for Invalid Encoded Strings)

**196. Problem: Implement error handling for cases where the encoded string might be invalid or corrupted (e.g., missing length prefix, invalid characters, etc.).**

**Example:**

```
Input: encoded = "5#hello4#worl"
Output: Error: Invalid encoded string
```

```
class Codec {
  encode(strs) {
    return strs.map((str) => `${str.length}#${str}`).join("");
  }

  decode(s) {
    const result = [];
    let i = 0;

    while (i < s.length) {
      let j = i;
      while (j < s.length && s[j] !== "#") {
        if (isNaN(s[j])) {
          throw new Error(
            "Invalid encoded string: Length prefix contains non-numeric characters"
          );
        }
        j++;
      }

      if (j === s.length) {
        throw new Error("Invalid encoded string: Missing delimiter (#)");
      }

      const length = parseInt(s.slice(i, j));
      if (isNaN(length) || length < 0) {
        throw new Error("Invalid encoded string: Invalid length prefix");
      }

      const str = s.slice(j + 1, j + 1 + length);
      if (str.length !== length) {
        throw new Error("Invalid encoded string: String length mismatch");
      }
    }
  }
}
```

```

        result.push(str);
        i = j + 1 + length;
    }

    return result;
}

}

// Test
try {
    const codec = new Codec();
    const encoded = codec.encode(["hello", "world"]);
    console.log("Encoded:", encoded); // Output: "5#hello5#world"

    const decoded = codec.decode("5#hello4#worl"); // Invalid encoded string
    console.log("Decoded:", decoded);
} catch (err) {
    console.error(err.message); // Output: Error: Invalid encoded string
}

```

### **Explanation:**

1. **Objective:** Introduce error handling to detect and reject invalid encoded strings.
2. **Approach:**
  - o During decoding, validate the format of the encoded string at each step.
  - o Check for non-numeric characters in the length prefix, missing delimiters, or mismatched lengths.
3. **Steps for Decoding:**
  - o Traverse the encoded string and validate each prefix, delimiter, and substring.
  - o Throw an error if any part of the encoded string is invalid.
4. **Why it Works:** Robust error handling ensures the decoding mechanism can handle unexpected inputs gracefully.
5. **Time Complexity:** O(n) — Processes each character in the encoded string.
6. **Space Complexity:** O(n) — Space for the decoded strings.

### **Encoding & Decoding Text-Based Data (Support Mixed Data Types)**

**197. Problem: Extend the encoding and decoding mechanism to handle a list of mixed data types such as strings, integers, and boolean values.**

### **Example:**

```

Input: data = ["hello", 42, true];
Encoded: "5#hello2#42#1#true";
Decoded: ["hello", "42", "true"];

```

**Solution:**

```
class Codec {
  encode(data) {
    return data
      .map((item) => {
        const str = String(item); // Convert all items to string
        return `${str.length}#${str}`;
      })
      .join("");
  }

  decode(s) {
    const result = [];
    let i = 0;

    while (i < s.length) {
      let j = i;
      while (s[j] !== "#" && j < s.length) {
        j++;
      }

      if (j === s.length) {
        throw new Error("Invalid encoded string: Missing delimiter (#)");
      }

      const length = parseInt(s.slice(i, j));
      if (isNaN(length) || length < 0) {
        throw new Error("Invalid encoded string: Invalid length prefix");
      }

      const str = s.slice(j + 1, j + 1 + length);
      result.push(str);

      i = j + 1 + length;
    }
  }

  return result;
}

// Test
const codec = new Codec();
const encoded = codec.encode(["hello", 42, true]);
console.log("Encoded:", encoded); // Output: "5#hello2#424#true"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["hello", "42", "true"]
```

### **Explanation:**

1. **Objective:** Support encoding and decoding of mixed data types by treating all data as strings during the process.
2. **Approach:**
  - o During **encoding**, convert all items to strings before applying the length-prefix encoding.
  - o During **decoding**, treat all decoded items as strings. (Data type restoration, if needed, can be handled separately.)
3. **Steps for Encoding:**
  - o Iterate through the list of items and convert each to a string.
  - o Apply the length-prefix encoding for each string.
  - o Concatenate all encoded strings into a single result.
4. **Steps for Decoding:**
  - o Parse the length prefix and extract the substring for each encoded segment.
  - o Treat all decoded substrings as strings.
5. **Why it Works:** Converting all data types to strings ensures compatibility with the length-prefix encoding mechanism.
6. **Time Complexity:**
  - o **Encoding:**  $O(n)$  — Processes each item in the list.
  - o **Decoding:**  $O(n)$  — Processes each character in the encoded string.
7. **Space Complexity:**  $O(n)$  — Space for the encoded/decoded string.

### **Encoding & Decoding Text-Based Data (Include Data Type Information)**

**198. Problem: Modify the encoding mechanism to include data type information for each item, allowing the decoded output to restore the original data types.**

#### **Example:**

```
Input: data = ["hello", 42, true];
Encoded: "S5#helloI2#42B1#true";
Decoded: ["hello", 42, true];
```

#### **Solution:**

```
class Codec {
  encode(data) {
    return data
      .map((item) => {
        const type =
          typeof item === "string" ? "S" : typeof item === "number" ? "I" : "B";
        const str = String(item);
        return `${type}${str.length}${str}`;
      })
      .join("");
  }

  decode(s) {
```

```

const result = [];
let i = 0;

while (i < s.length) {
  const type = s[i]; // Get the data type prefix
  i++;

  let j = i;
  while (s[j] !== "#" && j < s.length) {
    j++;
  }

  if (j === s.length) {
    throw new Error("Invalid encoded string: Missing delimiter (#)");
  }

  const length = parseInt(s.slice(i, j));
  if (isNaN(length) || length < 0) {
    throw new Error("Invalid encoded string: Invalid length prefix");
  }

  const str = s.slice(j + 1, j + 1 + length);
  if (type === "I") {
    result.push(Number(str)); // Restore numbers
  } else if (type === "B") {
    result.push(str === "true"); // Restore booleans
  } else {
    result.push(str); // Restore strings
  }

  i = j + 1 + length;
}

return result;
}

// Test
const codec = new Codec();
const encoded = codec.encode(["hello", 42, true]);
console.log("Encoded:", encoded); // Output: "S5#helloI2#42B1#true"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["hello", 42, true]

```

### Explanation:

- Objective:** Add type information to the encoding mechanism to support restoration of original data types during decoding.

## 2. Approach:

- During **encoding**, prefix each item with a type identifier:
  - S for strings.
  - I for integers.
  - B for booleans.
- During **decoding**, parse the type identifier and convert the decoded string back to its original data type.

## 3. Steps for Encoding:

- Determine the type of each item and prefix it with its corresponding identifier.
- Encode the string representation of the item using length-prefix encoding.

## 4. Steps for Decoding:

- Read the type identifier to determine how to convert the decoded substring.
- Parse the length prefix and extract the substring.
- Convert the substring back to its original data type based on the type identifier.

## 5. Why it Works:

Including type information ensures that all data types can be restored accurately after decoding.

## 6. Time Complexity:

- **Encoding:**  $O(n)$  — Processes each item in the list.
- **Decoding:**  $O(n)$  — Processes each character in the encoded string.

## 7. Space Complexity:

$O(n)$  — Space for the encoded/decoded string.

### Encoding & Decoding Text-Based Data (Nested Lists of Strings)

**199. Problem:** Support encoding and decoding of nested lists of strings, where the input can contain sublists that also need to be encoded and decoded.

#### Example:

```
Input: nestedStrings = ["hello", ["world", "nested"], "example"];
Encoded: "5#helloL11#5#world6#nested7#example";
Decoded: ["hello", ["world", "nested"], "example"];
```

#### Solution:

```
class Codec {
  encode(data) {
    return data
      .map((item) => {
        if (Array.isArray(item)) {
          const encodedList = this.encode(item); // Recursively encode sublists
          return `L${encodedList.length}#${encodedList}`;
        } else {
          return `${item.length}#${item}`;
        }
      })
      .join("");
  }

  decode(s) {
```

```

const result = [];
let i = 0;

while (i < s.length) {
  if (s[i] === "L") {
    // Handle nested list
    i++;
    let j = i;
    while (s[j] !== "#" && j < s.length) {
      j++;
    }

    if (j === s.length) {
      throw new Error("Invalid encoded string: Missing delimiter (#)");
    }

    const length = parseInt(s.slice(i, j));
    const nestedEncoded = s.slice(j + 1, j + 1 + length);
    result.push(this.decode(nestedEncoded)); // Recursively decode the sublist

    i = j + 1 + length;
  } else {
    // Handle regular string
    let j = i;
    while (s[j] !== "#" && j < s.length) {
      j++;
    }

    if (j === s.length) {
      throw new Error("Invalid encoded string: Missing delimiter (#)");
    }

    const length = parseInt(s.slice(i, j));
    const str = s.slice(j + 1, j + 1 + length);
    result.push(str);

    i = j + 1 + length;
  }
}

return result;
}

// Test
const codec = new Codec();
const nestedStrings = ["hello", ["world", "nested"], "example"];
const encoded = codec.encode(nestedStrings);
console.log("Encoded:", encoded); // Output: "5#helloL11#5#world6#nested7#example"

```

```
const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["hello", ["world", "nested"], "example"]
```

### Explanation:

1. **Objective:** Extend the encoding and decoding mechanism to handle nested lists of strings.
2. **Approach:**
  - o During **encoding**, identify whether an item is a sublist and recursively encode it with a prefix (L) to indicate a list.
  - o During **decoding**, handle the list prefix (L) by extracting the nested encoded string and recursively decoding it.
3. **Steps for Encoding:**
  - o For each item:
    - If it is a sublist, recursively encode it and prefix it with its length and L.
    - Otherwise, encode it as a regular string using length-prefix encoding.
4. **Steps for Decoding:**
  - o Traverse the encoded string:
    - If a list prefix (L) is encountered, extract and recursively decode the nested string.
    - Otherwise, decode the string as usual.
5. **Why it Works:** Recursive encoding and decoding ensure that all levels of nested lists are processed correctly without ambiguity.
6. **Time Complexity:**
  - o **Encoding:** O(n) — Each item in the input structure is processed once.
  - o **Decoding:** O(n) — Each character in the encoded string is processed once.
7. **Space Complexity:** O(n) — Space for the encoded/decoded string, including recursion stack for nested lists.

### Encoding & Decoding Text-Based Data (Handle Arbitrary Delimiters)

**200. Problem: Design an encoding mechanism that can use arbitrary delimiters and allow decoding with the same delimiter.**

### Example:

```
Input: (strings = ["hello", "world"]), (delimiter = "|");
Encoded: "5|hello5|world";
Decoded: ["hello", "world"];
```

### Solution:

```
class Codec {
  constructor(delimiter = "#") {
    this.delimiter = delimiter; // Custom delimiter
  }
}
```

```

encode(strs) {
    return strs.map((str) => `${str.length}${this.delimiter}${str}`).join("");
}

decode(s) {
    const result = [];
    let i = 0;

    while (i < s.length) {
        let j = i;
        while (s[j] !== this.delimiter && j < s.length) {
            j++;
        }

        if (j === s.length) {
            throw new Error("Invalid encoded string: Missing delimiter");
        }

        const length = parseInt(s.slice(i, j));
        const str = s.slice(j + 1, j + 1 + length);
        result.push(str);

        i = j + 1 + length;
    }

    return result;
}
}

// Test
const codec = new Codec("|");
const strings = ["hello", "world"];
const encoded = codec.encode(strings);
console.log("Encoded:", encoded); // Output: "5|hello5|world"

const decoded = codec.decode(encoded);
console.log("Decoded:", decoded); // Output: ["hello", "world"]

```

### Explanation:

1. **Objective:** Allow custom delimiters in the encoding and decoding mechanism for flexibility.
2. **Approach:**
  - o Pass the desired delimiter to the Codec class during initialization.
  - o Replace the hardcoded delimiter (#) with the custom delimiter in both encoding and decoding.
3. **Steps for Encoding:**
  - o Concatenate each string with its length and the custom delimiter.
4. **Steps for Decoding:**
  - o Parse the length prefix using the custom delimiter and extract the substring.

5. **Why it Works:** Parameterizing the delimiter ensures compatibility with a variety of input formats and allows customization.
6. **Time Complexity:**
  - o **Encoding:**  $O(n)$  — Processes each character in the input strings.
  - o **Decoding:**  $O(n)$  — Processes each character in the encoded string.
7. **Space Complexity:**  $O(n)$  — Space for the encoded/decoded string.

### 3. Trees & Hierarchical Data Structures

#### Determining Tree Depth & Size (Recursive Approach)

**201. Problem:** Given the root of a binary tree, return its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example:**

Input: root = [3, 9, 20, null, null, 15, 7]

Output: 3

Explanation: The maximum depth is 3 because the longest path is 3 → 20 → 15 or 3 → 20 → 7.

**Solution:**

```
function maxDepth(root) {  
    if (root === null) {  
        return 0;  
    }  
    const leftDepth = maxDepth(root.left);  
    const rightDepth = maxDepth(root.right);  
    return Math.max(leftDepth, rightDepth) + 1;  
}  
  
// Test  
const tree = {  
    val: 3,  
    left: { val: 9, left: null, right: null },  
    right: {  
        val: 20,  
        left: { val: 15, left: null, right: null },  
        right: { val: 7, left: null, right: null },  
    },  
};  
console.log(maxDepth(tree)); // Output: 3
```

**Explanation:**

1. **Objective:** Find the longest path from the root node to any leaf node.

2. **Approach:**

- Use recursion to calculate the depth of the left and right subtrees.
- For each node, the maximum depth is the greater of the left and right subtree depths plus 1 (for the current node).

3. **Steps:**

- If the root is null, return 0 because an empty tree has a depth of 0.
- Recursively calculate the depth of the left and right subtrees.

- Return the maximum of the two depths plus 1 to account for the current node.
4. **Why it Works:** The recursive approach ensures that the depth of every subtree is calculated and compared, guaranteeing the maximum depth is found.
  5. **Time Complexity:**  $O(n)$  — Every node is visited once.
  6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Determining Tree Depth & Size (Iterative Approach using Level Order Traversal)

**202. Problem: Find the maximum depth of a binary tree using an iterative approach with level-order traversal.**

**Example:**

Input: root = [3, 9, 20, null, null, 15, 7]

Output: 3

Explanation: The maximum depth is 3 because the longest path is  $3 \rightarrow 20 \rightarrow 15$  or  $3 \rightarrow 20 \rightarrow 7$ .

**Solution:**

```
function maxDepth(root) {
  if (root === null) {
    return 0;
  }

  const queue = [root];
  let depth = 0;

  while (queue.length > 0) {
    const levelSize = queue.length;

    for (let i = 0; i < levelSize; i++) {
      const currentNode = queue.shift();

      if (currentNode.left !== null) {
        queue.push(currentNode.left);
      }
      if (currentNode.right !== null) {
        queue.push(currentNode.right);
      }
    }

    depth++;
  }

  return depth;
}
```

```
// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};
console.log(maxDepth(tree)); // Output: 3
```

### Explanation:

1. **Objective:** Use an iterative approach to calculate the maximum depth of the binary tree by traversing it level by level.
2. **Approach:**
  - o Use a queue for level-order traversal (breadth-first traversal).
  - o For each level of the tree, increment the depth counter.
3. **Steps:**
  - o Initialize a queue with the root node.
  - o While the queue is not empty:
    - Count the nodes in the current level (levelSize).
    - Iterate over all nodes in the current level, adding their children to the queue for the next level.
    - Increment the depth counter after processing each level.
4. **Why it Works:** Each iteration of the outer loop processes one level of the tree, and the depth counter tracks the number of levels traversed.
5. **Time Complexity:**  $O(n)$  — Every node is visited once.
6. **Space Complexity:**  $O(n)$  — The queue can store up to  $n$  nodes in the worst case (for a completely balanced tree).

### Determining Tree Depth & Size (DFS with Stack)

**203. Problem:** **Find the maximum depth of a binary tree using an iterative depth-first search approach with a stack.**

#### Example:

```
Input: root = [3, 9, 20, null, null, 15, 7];
Output: 3;
```

#### Solution:

```
function maxDepth(root) {
  if (root === null) {
    return 0;
  }
```

```

const stack = [{ node: root, depth: 1 }];
let maxDepth = 0;

while (stack.length > 0) {
  const { node, depth } = stack.pop();

  if (node !== null) {
    maxDepth = Math.max(maxDepth, depth);

    if (node.left !== null) {
      stack.push({ node: node.left, depth: depth + 1 });
    }
    if (node.right !== null) {
      stack.push({ node: node.right, depth: depth + 1 });
    }
  }
}

return maxDepth;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};
console.log(maxDepth(tree)); // Output: 3

```

### Explanation:

1. **Objective:** Use an iterative depth-first search (DFS) to calculate the maximum depth of the binary tree.
2. **Approach:**
  - o Use a stack to simulate the recursive depth-first traversal of the tree.
  - o Store each node and its depth in the stack.
  - o Track the maximum depth encountered during traversal.
3. **Steps:**
  - o Initialize a stack with the root node and its depth (1).
  - o While the stack is not empty:
    - Pop the top element and update the maxDepth if the current depth is greater.
    - Push the left and right children (if any) onto the stack with their corresponding depths.

- Return the maxDepth after traversal.
4. **Why it Works:** The stack ensures that each node is visited once, and the depth is updated dynamically as nodes are processed.
  5. **Time Complexity:**  $O(n)$  — Every node is visited once.
  6. **Space Complexity:**  $O(h)$  — The stack stores nodes up to the height of the tree.

### **Determining Tree Depth & Size (Divide and Conquer Approach)**

**204. Problem:** **Find the maximum depth of a binary tree using a divide-and-conquer method.**

**Example:**

Input: root = [3, 9, 20, null, null, 15, 7]

Output: 3

Explanation: The maximum depth is 3 because the longest path is  $3 \rightarrow 20 \rightarrow 15$  or  $3 \rightarrow 20 \rightarrow 7$ .

**Solution:**

```
function maxDepth(root) {
  if (root === null) {
    return 0;
  }

  const leftDepth = maxDepth(root.left);
  const rightDepth = maxDepth(root.right);

  return Math.max(leftDepth, rightDepth) + 1;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};
console.log(maxDepth(tree)); // Output: 3
```

**Explanation:**

1. **Objective:** Calculate the maximum depth of the binary tree using a divide-and-conquer strategy.
2. **Approach:**

- Break down the problem into smaller subproblems by calculating the maximum depth of the left and right subtrees.
  - Combine the results by taking the maximum of the two depths and adding 1 to account for the current node.
3. **Steps:**
- Base Case: If the current node is null, return a depth of 0.
  - Recursively calculate the depth of the left and right subtrees.
  - Return the maximum of the left and right depths plus 1.
4. **Why it Works:** The divide-and-conquer approach ensures that the depth of each subtree is calculated independently before combining the results.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### **Determining Tree Depth & Size (Morris Traversal for Constant Space)**

**205. Problem: Find the maximum depth of a binary tree using Morris Traversal, which achieves O(1) space complexity.**

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7];
Output: 3;
```

**Solution:**

```
function maxDepth(root) {
  if (root === null) return 0;

  let current = root;
  let depth = 0;
  let maxDepth = 0;

  while (current !== null) {
    if (current.left === null) {
      depth++;
      maxDepth = Math.max(maxDepth, depth);
      current = current.right;
    } else {
      let predecessor = current.left;
      let steps = 1;

      while (predecessor.right !== null && predecessor.right !== current) {
        predecessor = predecessor.right;
        steps++;
      }

      if (predecessor.right === null) {
        predecessor.right = current;
        depth--;
      }
    }
  }
}
```

```

        depth++;
        current = current.left;
    } else {
        predecessor.right = null;
        depth -= steps;
        current = current.right;
    }

    maxDepth = Math.max(maxDepth, depth);
}
}

return maxDepth;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};
console.log(maxDepth(tree)); // Output: 3

```

### Explanation:

1. **Objective:** Use Morris Traversal to calculate the maximum depth of a binary tree without additional space for a stack or queue.
2. **Approach:**
  - o Modify the tree temporarily to create links to in-order predecessors (Morris Traversal).
  - o Use these links to traverse the tree iteratively while keeping track of the depth.
3. **Steps:**
  - o For each node, check if it has a left child:
    - If no left child, move to the right child and increment the depth.
    - If a left child exists, find the in-order predecessor:
      - If the predecessor's right pointer is null, create a temporary link to the current node and move left.
      - If the predecessor's right pointer is already linked, remove the link, decrement the depth, and move right.
  - o Update the maximum depth encountered during the traversal.
4. **Why it Works:** Morris Traversal modifies the tree temporarily, allowing traversal without using extra space.
5. **Time Complexity:**  $O(n)$  — Each node is visited a constant number of times.
6. **Space Complexity:**  $O(1)$  — No additional space is used apart from variables.

## Determining Tree Depth & Size (Bottom-Up Recursion)

**206. Problem:** Find the maximum depth of a binary tree using a bottom-up recursive approach.

**Example:**

Input: root = [3, 9, 20, null, null, 15, 7]

Output: 3

Explanation: The longest path is from the root (3) to a leaf (20 → 15 or 20 → 7).

**Solution:**

```
function maxDepth(root) {  
    if (root === null) {  
        return 0;  
    }  
  
    const leftDepth = maxDepth(root.left);  
    const rightDepth = maxDepth(root.right);  
  
    return Math.max(leftDepth, rightDepth) + 1;  
}  
  
// Test  
const tree = {  
    val: 3,  
    left: { val: 9, left: null, right: null },  
    right: {  
        val: 20,  
        left: { val: 15, left: null, right: null },  
        right: { val: 7, left: null, right: null },  
    },  
};  
console.log(maxDepth(tree)); // Output: 3
```

**Explanation:**

1. **Objective:** Use a bottom-up recursive approach to calculate the depth of the binary tree starting from the leaf nodes.
2. **Approach:**
  - Start the recursion at the root node.
  - For each node, calculate the depth of its left and right subtrees.
  - Combine the results by returning the maximum depth of the two subtrees plus 1 (for the current node).
3. **Steps:**
  - Base Case: If the current node is null, return 0 (leaf node reached).

- Recursive Case: Calculate the depth of the left and right subtrees, then return the maximum of the two depths plus 1.
4. **Why it Works:** This approach ensures that the depth of each subtree is fully calculated before combining the results.
  5. **Time Complexity:**  $O(n)$  — Each node is visited once.
  6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Determining Tree Depth & Size (Top-Down Recursion)

**207. Problem: Calculate the maximum depth of a binary tree using a top-down recursion approach, where depth is passed as a parameter during the recursive calls.**

**Solution:**

```
function maxDepth(root) {
  function calculateDepth(node, currentDepth) {
    if (node === null) {
      return currentDepth;
    }

    const leftDepth = calculateDepth(node.left, currentDepth + 1);
    const rightDepth = calculateDepth(node.right, currentDepth + 1);

    return Math.max(leftDepth, rightDepth);
  }

  return calculateDepth(root, 0);
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};
console.log(maxDepth(tree)); // Output: 3
```

**Explanation:**

1. **Objective:** Use a top-down recursive approach to calculate the depth by maintaining a depth counter during recursive calls.
2. **Approach:**
  - Pass the current depth as a parameter to each recursive call.
  - At each leaf node, return the current depth.

- The maximum depth is calculated by comparing the depths returned from the left and right subtrees.
3. **Steps:**
- Start the recursion at the root with an initial depth of 0.
  - For each node, increment the depth counter and pass it to the recursive calls for the left and right children.
  - Return the maximum depth encountered during the traversal.
4. **Why it Works:** By maintaining the depth as a parameter, this approach avoids the need to recompute depth during the recursion.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Determining Tree Depth & Size (Using Post-Order Traversal)

**208. Problem: Find the maximum depth of a binary tree using post-order traversal.**

**Solution:**

```
function maxDepth(root) {
  if (root === null) {
    return 0;
  }

  const leftDepth = maxDepth(root.left);
  const rightDepth = maxDepth(root.right);

  return Math.max(leftDepth, rightDepth) + 1;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};
console.log(maxDepth(tree)); // Output: 3
```

### **Detailed Explanation:**

1. **Objective:** Use post-order traversal to calculate the depth of a binary tree.
2. **Approach:**
  - Post-order traversal ensures that the left and right subtrees are processed before combining their results.

- For each node, the depth is calculated by adding 1 to the maximum depth of its left and right subtrees.
3. **Steps:**
- Traverse the left and right subtrees recursively.
  - Combine the results by taking the maximum depth of the two subtrees and adding 1.
4. **Why it Works:** Post-order traversal ensures that all child nodes are processed before calculating the depth for the current node.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### **Determining Tree Depth & Size (Using BFS with Depth Counter)**

**209. Problem: Find the maximum depth of a binary tree using Breadth-First Search (BFS) with a depth counter.**

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7];
Output: 3;
```

**Solution:**

```
function maxDepth(root) {
  if (root === null) {
    return 0;
  }

  const queue = [root];
  let depth = 0;

  while (queue.length > 0) {
    let levelSize = queue.length;

    while (levelSize > 0) {
      const currentNode = queue.shift();

      if (currentNode.left) {
        queue.push(currentNode.left);
      }

      if (currentNode.right) {
        queue.push(currentNode.right);
      }

      levelSize--;
    }
  }
}
```

```

    depth++;
}

return depth;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};
console.log(maxDepth(tree)); // Output: 3

```

### **Explanation:**

1. **Objective:** Traverse the binary tree level by level, counting the number of levels to find the maximum depth.
2. **Approach:**
  - o Use a queue to store nodes level by level during traversal.
  - o Increment the depth counter after processing all nodes in the current level.
3. **Steps:**
  - o Initialize the queue with the root node.
  - o While the queue is not empty:
    - Process all nodes in the current level (determine levelSize).
    - Add the children of the current level's nodes to the queue.
    - Increment the depth counter.
  - o Return the depth counter after traversal.
4. **Why it Works:** BFS processes nodes level by level, and the depth counter tracks the number of levels traversed.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — The queue can hold up to  $n$  nodes in the worst case (for a completely balanced tree).

### **Determining Tree Depth & Size (With Null Markers in BFS)**

**210. Problem:** Use a null marker to indicate the end of each level during BFS and find the maximum depth.

### **Solution:**

```

function maxDepth(root) {
  if (root === null) {
    return 0;
  }
}

```

```

const queue = [root, null];
let depth = 0;

while (queue.length > 0) {
  const currentNode = queue.shift();

  if (currentNode === null) {
    depth++;
    if (queue.length > 0) {
      queue.push(null);
    }
  } else {
    if (currentNode.left) {
      queue.push(currentNode.left);
    }

    if (currentNode.right) {
      queue.push(currentNode.right);
    }
  }
}

return depth;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};
console.log(maxDepth(tree)); // Output: 3

```

### Explanation:

1. **Objective:** Use a null marker to separate levels during BFS and count the number of levels to determine the maximum depth.
2. **Approach:**
  - o Add a null marker to the queue after processing each level.
  - o Increment the depth counter whenever a null marker is encountered.
3. **Steps:**
  - o Initialize the queue with the root node and a null marker.
  - o Traverse the queue:

- If a null marker is encountered, increment the depth counter and add another null marker if more nodes remain.
  - Otherwise, process the current node and add its children to the queue.
- Return the depth counter after traversal.
4. **Why it Works:** The null marker separates levels, and the depth counter tracks the number of null markers encountered.
  5. **Time Complexity:**  $O(n)$  — Each node is visited once.
  6. **Space Complexity:**  $O(n)$  — The queue can hold up to  $n$  nodes in the worst case.

### Comparing Two Tree Structures (Recursive Approach)

**211. Problem:** Given the roots of two binary trees,  $p$  and  $q$ , determine if they are the same.

Two binary trees are considered the same if they are structurally identical and the nodes have the same values.

**Example:**

Input: ( $p = [1, 2, 3]$ ), ( $q = [1, 2, 3]$ );  
Output: **true**;

Input: ( $p = [1, 2]$ ), ( $q = [1, \text{null}, 2]$ );  
Output: **false**;

**Solution:**

```
function isSameTree(p, q) {
  if (p === null && q === null) {
    return true;
  }
  if (p === null || q === null || p.val !== q.val) {
    return false;
  }

  return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}

// Test
const tree1 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};

const tree2 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};
```

```

console.log(isSameTree(tree1, tree2)); // Output: true

const tree3 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: null,
};

console.log(isSameTree(tree1, tree3)); // Output: false

```

### **Explanation:**

1. **Objective:** Determine if two trees are identical by recursively comparing their structures and values.
2. **Approach:**
  - o Use a recursive function to compare nodes in the same positions in both trees.
  - o If both nodes are null, they are the same.
  - o If only one node is null or their values differ, the trees are not the same.
  - o Recursively check the left and right subtrees.
3. **Steps:**
  - o Base Case: If both nodes are null, return true.
  - o If only one node is null or their values are different, return false.
  - o Recursively check the left and right subtrees of the current nodes.
4. **Why it Works:** The recursive approach ensures that all corresponding nodes in both trees are compared for equality.
5. **Time Complexity:**  $O(n)$  — Each node in both trees is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### **Comparing Two Tree Structures (Iterative Approach Using Queue)**

**212. Problem:** Given the roots of two binary trees,  $p$  and  $q$ , determine if they are the same using an iterative approach.

#### **Example:**

Input: ( $p = [1, 2, 3]$ ), ( $q = [1, 2, 3]$ );  
Output: **true**;

Input: ( $p = [1, 2]$ ), ( $q = [1, \text{null}, 2]$ );  
Output: **false**;

#### **Solution:**

```

function isSameTree(p, q) {
  const queue = [[p, q]];
  while (queue.length > 0) {
    const [currentP, currentQ] = queue.shift();
  }
}

```

```

const [node1, node2] = queue.shift();

if (node1 === null && node2 === null) {
    continue;
}

if (node1 === null || node2 === null || node1.val !== node2.val) {
    return false;
}

queue.push([node1.left, node2.left]);
queue.push([node1.right, node2.right]);
}

return true;
}

// Test
const tree1 = {
    val: 1,
    left: { val: 2, left: null, right: null },
    right: { val: 3, left: null, right: null },
};

const tree2 = {
    val: 1,
    left: { val: 2, left: null, right: null },
    right: { val: 3, left: null, right: null },
};

console.log(isSameTree(tree1, tree2)); // Output: true

const tree3 = {
    val: 1,
    left: { val: 2, left: null, right: null },
    right: null,
};

console.log(isSameTree(tree1, tree3)); // Output: false

```

### Explanation:

1. **Objective:** Use an iterative approach to determine if two trees are identical by comparing corresponding nodes in a level-order fashion.
2. **Approach:**
  - o Use a queue to store pairs of nodes to be compared.
  - o Dequeue and compare the nodes:
    - If both are null, continue to the next pair.
    - If one is null or their values differ, return false.

- Otherwise, enqueue their left and right children for further comparison.
3. **Steps:**
- Initialize the queue with a pair of root nodes ([p, q]).
  - While the queue is not empty:
    - Dequeue a pair of nodes and compare them.
    - If they are valid, enqueue their left and right children.
  - If all nodes are processed without mismatches, return true.
4. **Why it Works:** The iterative approach ensures all corresponding nodes in the two trees are compared systematically.
5. **Time Complexity:** O(n) — Each node in both trees is visited once.
6. **Space Complexity:** O(n) — The queue can hold up to n nodes in the worst case.

### **Comparing Two Tree Structures (Iterative Approach Using Stack)**

**213. Problem: Determine if two binary trees are the same using a depth-first traversal approach with a stack.**

**Solution:**

```
function isSameTree(p, q) {
  const stack = [[p, q]];

  while (stack.length > 0) {
    const [node1, node2] = stack.pop();

    if (node1 === null && node2 === null) {
      continue;
    }

    if (node1 === null || node2 === null || node1.val !== node2.val) {
      return false;
    }

    stack.push([node1.right, node2.right]);
    stack.push([node1.left, node2.left]);
  }

  return true;
}

// Test
const tree1 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};

const tree2 = {
  val: 1,
  left: { val: 2, left: null, right: null },
}
```

```

right: { val: 3, left: null, right: null },
};

console.log(isSameTree(tree1, tree2)); // Output: true

const tree3 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: null,
};

console.log(isSameTree(tree1, tree3)); // Output: false

```

### Explanation:

1. **Objective:** Use a stack to perform a depth-first traversal of both trees and compare corresponding nodes.
2. **Approach:**
  - o Push pairs of nodes ([p, q]) onto the stack for comparison.
  - o Pop pairs from the stack and compare:
    - If both are null, continue to the next pair.
    - If one is null or their values differ, return false.
    - Otherwise, push their left and right children onto the stack.
3. **Steps:**
  - o Initialize the stack with the root nodes ([p, q]).
  - o While the stack is not empty:
    - Pop a pair of nodes and compare them.
    - If they are valid, push their left and right children onto the stack.
  - o If all nodes are processed without mismatches, return true.
4. **Why it Works:** The stack ensures that nodes are processed in a depth-first manner, and all corresponding nodes are compared.
5. **Time Complexity:** O(n) — Each node in both trees is visited once.
6. **Space Complexity:** O(h) — The stack stores nodes up to the height of the tree.

### Comparing Two Tree Structures (Check if One Tree is a Subtree of Another)

**214. Problem:** Given two binary trees, p and q, determine if p is the same as q or if p is a subtree of q.

#### Example:

Input: (p = [4, 1, 2]), (q = [3, 4, 5, 1, 2]);  
Output: true;

Input: (p = [4, 1, 2]), (q = [3, 4, 5, 1, null, 2]);  
Output: false;

### Solution:

```
function isSameTree(p, q) {  
    if (p === null && q === null) {  
        return true;  
    }  
    if (p === null || q === null || p.val !== q.val) {  
        return false;  
    }  
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);  
}  
  
function isSubtree(q, p) {  
    if (q === null) {  
        return false;  
    }  
    if (isSameTree(q, p)) {  
        return true;  
    }  
    return isSubtree(q.left, p) || isSubtree(q.right, p);  
}  
  
// Test  
const tree1 = {  
    val: 4,  
    left: { val: 1, left: null, right: null },  
    right: { val: 2, left: null, right: null },  
};  
  
const tree2 = {  
    val: 3,  
    left: {  
        val: 4,  
        left: { val: 1, left: null, right: null },  
        right: { val: 2, left: null, right: null },  
    },  
    right: { val: 5, left: null, right: null },  
};  
  
console.log(isSubtree(tree2, tree1)); // Output: true
```

### Explanation:

1. **Objective:** Check if one tree is the same as another or is a Checking Subtree Presence.
2. **Approach:**
  - o Use the recursive isSameTree function to compare nodes and subtrees.
  - o Traverse the larger tree (q) using a recursive function (isSubtree) and check if any subtree matches the smaller tree (p).

3. **Steps:**
  - o Base Case: If the larger tree (q) is null, return false since p cannot be a subtree of an empty tree.
  - o If the current nodes of both trees match (`isSameTree(q, p)`), return true.
  - o Otherwise, recursively check the left and right subtrees of q to see if p matches any subtree.
4. **Why it Works:** This approach ensures that every subtree of q is compared against p using the `isSameTree` function.
5. **Time Complexity:**  $O(m * n)$  — For every node in q (m nodes), compare with all nodes in p (n nodes).
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where h is the height of q.

### **Comparing Two Tree Structures (Check Symmetry Between Two Trees)**

**215. Problem: Check if two binary trees are mirror images of each other.**

**Example:**

```
Input: (p = [1, 2, 2, 3, 4, 4, 3]), (q = [1, 2, 2, 4, 3, 3, 4]);
Output: true;
```

**Solution:**

```
function isSymmetricTree(p, q) {
  if (p === null && q === null) {
    return true;
  }
  if (p === null || q === null || p.val !== q.val) {
    return false;
  }

  return isSymmetricTree(p.left, q.right) && isSymmetricTree(p.right, q.left);
}

// Test
const tree1 = {
  val: 1,
  left: {
    val: 2,
    left: { val: 3, left: null, right: null },
    right: { val: 4, left: null, right: null },
  },
  right: {
    val: 2,
    left: { val: 4, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
};
```

```

const tree2 = {
    val: 1,
    left: {
        val: 2,
        left: { val: 4, left: null, right: null },
        right: { val: 3, left: null, right: null },
    },
    right: {
        val: 2,
        left: { val: 3, left: null, right: null },
        right: { val: 4, left: null, right: null },
    },
};

console.log(isSymmetricTree(tree1, tree2)); // Output: true

```

### Explanation:

1. **Objective:** Check if two trees are symmetrical, meaning one is a mirror image of the other.
2. **Approach:**
  - o Use a recursive function to compare nodes in a symmetric fashion.
  - o The left subtree of the first tree is compared to the right subtree of the second tree, and vice versa.
3. **Steps:**
  - o Base Case: If both nodes are null, return true.
  - o If only one node is null or their values differ, return false.
  - o Recursively compare the left subtree of the first tree with the right subtree of the second tree, and vice versa.
4. **Why it Works:** The symmetric comparison ensures that all nodes are mirrored correctly between the two trees.
5. **Time Complexity:**  $O(n)$  — Each node in both trees is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the trees.

### Comparing Two Tree Structures (Check if Two Trees are Identical at a Given Level)

**216. Problem:** Given the roots of two binary trees, p and q, check if the nodes at a specific level k are identical.

### Example:

Input:  $p = [1, 2, 3, 4, 5]$ ,  $q = [1, 2, 3, 4, 6]$ ,  $k = 3$

Output: `false`

Explanation: At level 3, the nodes in tree `p` are [4, 5], and in tree `q` are [4, 6], which are not identical.

### Solution:

```
function isSameLevel(p, q, k) {
    if (p === null && q === null) {
        return true;
    }
    if (p === null || q === null) {
        return false;
    }
    if (k === 1) {
        return p.val === q.val;
    }
    return (
        isSameLevel(p.left, q.left, k - 1) && isSameLevel(p.right, q.right, k - 1)
    );
}

// Test
const tree1 = {
    val: 1,
    left: {
        val: 2,
        left: { val: 4, left: null, right: null },
        right: { val: 5, left: null, right: null },
    },
    right: { val: 3, left: null, right: null },
};

const tree2 = {
    val: 1,
    left: {
        val: 2,
        left: { val: 4, left: null, right: null },
        right: { val: 6, left: null, right: null },
    },
    right: { val: 3, left: null, right: null },
};

console.log(isSameLevel(tree1, tree2, 3)); // Output: false
console.log(isSameLevel(tree1, tree2, 2)); // Output: true
```

### Explanation:

1. **Objective:** Check if all nodes at a specific level k in two trees are identical.
2. **Approach:**
  - o Use recursion to traverse the trees and decrement the level k at each step.
  - o When k reaches 1, compare the values of the nodes at that level.
3. **Steps:**
  - o Base Case:

- If both nodes are null, return true because they are identical at this level.
  - If one node is null and the other is not, return false.
  - If  $k == 1$ , compare the values of the nodes and return true or false.
- Recursive Case: Traverse the left and right subtrees with  $k - 1$ .
4. **Why it Works:** By decrementing the level  $k$ , this approach ensures that nodes at the desired level are compared directly.
  5. **Time Complexity:**  $O(n)$  — Visits all nodes up to level  $k$  in both trees.
  6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the trees.

### Comparing Two Tree Structures (Check if Two Trees are Structurally Identical)

**217. Problem: Given two binary trees, determine if they are structurally identical, regardless of their node values.**

**Example:**

Input: ( $p = [1, 2, 3]$ ), ( $q = [4, 5, 6]$ );  
 Output: **true**;

Input: ( $p = [1, 2]$ ), ( $q = [1, \text{null}, 2]$ );  
 Output: **false**;

**Solution:**

```
function isStructurallyIdentical(p, q) {
  if (p === null && q === null) {
    return true;
  }
  if (p === null || q === null) {
    return false;
  }

  return (
    isStructurallyIdentical(p.left, q.left) &&
    isStructurallyIdentical(p.right, q.right)
  );
}

// Test
const tree1 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};

const tree2 = {
  val: 4,
```

```

left: { val: 5, left: null, right: null },
right: { val: 6, left: null, right: null },
};

const tree3 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: null,
};

console.log(isStructurallyIdentical(tree1, tree2)); // Output: true
console.log(isStructurallyIdentical(tree1, tree3)); // Output: false

```

### Explanation:

1. **Objective:** Check if two trees have the same structure, ignoring their node values.
2. **Approach:**
  - o Use recursion to compare the presence of left and right subtrees in both trees.
  - o If both nodes are null, they are structurally identical.
  - o If only one node is null or the structure of their subtrees differs, return false.
3. **Steps:**
  - o Base Case:
    - If both nodes are null, return true.
    - If only one node is null, return false.
  - o Recursive Case:
    - Check if the left subtrees are structurally identical.
    - Check if the right subtrees are structurally identical.
4. **Why it Works:** By recursively comparing the structure of left and right subtrees, this approach ensures that all nodes are aligned in the same way.
5. **Time Complexity:**  $O(n)$  — Each node in both trees is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the trees.

### Comparing Two Tree Structures (Check if Two Trees are Inverse)

**218. Problem:** Given the roots of two binary trees, p and q, determine if one tree is the inverse (mirror image) of the other.

### Example:

Input: ( $p = [1, 2, 3]$ ), ( $q = [1, 3, 2]$ );  
Output: **true**;

Input: ( $p = [1, 2]$ ), ( $q = [1, \text{null}, 2]$ );  
Output: **false**;

### Solution:

```

function isInverseTree(p, q) {
  if (p === null && q === null) {
    return true;
  }
  if (p === null || q === null || p.val !== q.val) {
    return false;
  }

  return isInverseTree(p.left, q.right) && isInverseTree(p.right, q.left);
}

// Test
const tree1 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};

const tree2 = {
  val: 1,
  left: { val: 3, left: null, right: null },
  right: { val: 2, left: null, right: null },
};

console.log(isInverseTree(tree1, tree2)); // Output: true

const tree3 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: null,
};

console.log(isInverseTree(tree1, tree3)); // Output: false

```

### Explanation:

1. **Objective:** Check if two binary trees are inverse (mirror image) of each other.
2. **Approach:**
  - o Use recursion to compare nodes at corresponding positions in the two trees.
  - o The left subtree of p should match the right subtree of q, and vice versa.
  - o The values of the nodes should also match.
3. **Steps:**
  - o Base Case:
    - If both nodes are null, return true because they are mirror images at this point.
    - If one node is null or their values are different, return false.
  - o Recursive Case:
    - Compare the left subtree of p with the right subtree of q.
    - Compare the right subtree of p with the left subtree of q.

4. **Why it Works:** This approach ensures that the structural symmetry and node values are matched at all levels.
5. **Time Complexity:**  $O(n)$  — Each node in both trees is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the trees.

### **Comparing Two Tree Structures (Iterative Check for Inverse Trees)**

**219. Problem: Determine if two binary trees are inverse of each other using an iterative approach.**

#### **Solution:**

```
function isInverseTreeIterative(p, q) {
  const stack = [[p, q]];

  while (stack.length > 0) {
    const [node1, node2] = stack.pop();

    if (node1 === null && node2 === null) {
      continue;
    }
    if (node1 === null || node2 === null || node1.val !== node2.val) {
      return false;
    }

    stack.push([node1.left, node2.right]);
    stack.push([node1.right, node2.left]);
  }

  return true;
}

// Test
const tree1 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};

const tree2 = {
  val: 1,
  left: { val: 3, left: null, right: null },
  right: { val: 2, left: null, right: null },
};

console.log(isInverseTreeIterative(tree1, tree2)); // Output: true

const tree3 = {
```

```

val: 1,
left: { val: 2, left: null, right: null },
right: null,
};

console.log(isInverseTreeIterative(tree1, tree3)); // Output: false

```

### Detailed Explanation:

1. **Objective:** Use an iterative approach to check if two binary trees are inverse of each other.
2. **Approach:**
  - o Use a stack to compare corresponding nodes in the two trees.
  - o For every pair of nodes:
    - If both are null, continue.
    - If one is null or their values differ, return false.
    - Push the left and right children of the nodes onto the stack in a mirrored fashion.
3. **Steps:**
  - o Initialize a stack with the root nodes of p and q.
  - o While the stack is not empty:
    - Pop a pair of nodes and compare them.
    - If they are valid, push their left and right children in a mirrored order onto the stack.
  - o If all nodes are processed without mismatches, return true.
4. **Why it Works:** The stack ensures that nodes are processed iteratively in a symmetric order.
5. **Time Complexity:**  $O(n)$  — Each node in both trees is visited once.
6. **Space Complexity:**  $O(h)$  — The stack stores nodes up to the height of the trees.

### Comparing Two Tree Structures (Flatten Trees and Compare)

**220. Problem: Flatten two trees into arrays using pre-order traversal and compare them to check if they are the same.**

### Solution:

```

function flattenTree(root) {
  if (root === null) {
    return ["null"];
  }

  return [root.val, ...flattenTree(root.left), ...flattenTree(root.right)];
}

function isSameTreeFlattened(p, q) {
  const pFlattened = flattenTree(p);
  const qFlattened = flattenTree(q);
}

```

```

    return JSON.stringify(pFlattened) === JSON.stringify(qFlattened);
}

// Test
const tree1 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};

const tree2 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 3, left: null, right: null },
};

const tree3 = {
  val: 1,
  left: { val: 2, left: null, right: null },
  right: { val: 4, left: null, right: null },
};

console.log(isSameTreeFlattened(tree1, tree2)); // Output: true
console.log(isSameTreeFlattened(tree1, tree3)); // Output: false

```

### **Explanation:**

1. **Objective:** Flatten both trees into arrays and compare their serialized forms.
2. **Approach:**
  - o Use pre-order traversal to serialize each tree into an array.
  - o Compare the serialized arrays using `JSON.stringify`.
3. **Steps:**
  - o Traverse both trees recursively.
  - o Store null for missing nodes to preserve the tree structure.
  - o Compare the resulting arrays.
4. **Why it Works:** Flattening the trees preserves their structure and values, making them easy to compare.
5. **Time Complexity:**  $O(n)$  — Each node is visited once during traversal.
6. **Space Complexity:**  $O(n)$  — Space for the flattened arrays.

### **Invert Binary Tree (Recursive Approach)**

**221. Problem:** Given the root of a binary tree, invert the tree (swap the left and right children for every node) and return its root.

### **Example:**

```
Input: root = [4, 2, 7, 1, 3, 6, 9]
Output: [4, 7, 2, 9, 6, 3, 1]
```

### Solution:

```
function invertTree(root) {
  if (root === null) {
    return null;
  }

  const left = invertTree(root.left);
  const right = invertTree(root.right);

  root.left = right;
  root.right = left;

  return root;
}

// Test
const tree = {
  val: 4,
  left: {
    val: 2,
    left: { val: 1, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
  right: {
    val: 7,
    left: { val: 6, left: null, right: null },
    right: { val: 9, left: null, right: null },
  },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{ "val":4,"left":{ "val":7,"left":{ "val":9,"left":null,"right":null },"right":{ "val":6,"left":null,"right":null } },"right":{ "val":2,"left":{ "val":3,"left":null,"right":null },"right":{ "val":1,"left":null,"right":null } }
```

### Explanation:

1. **Objective:** Swap the left and right children of all nodes in the binary tree.

2. **Approach:**

- Use recursion to invert the left and right subtrees first.
- After inverting the subtrees, swap them for the current node.

3. **Steps:**
  - o Base Case: If the current node is null, return null.
  - o Recursively invert the left and right subtrees.
  - o Swap the left and right subtrees for the current node.
  - o Return the root node.
4. **Why it Works:** Recursively swapping subtrees ensures that all levels of the tree are inverted correctly.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Invert Binary Tree (Iterative Approach Using Queue)

**222. Problem: Invert a binary tree using an iterative approach with a queue (Breadth-First Search).**

**Example:**

```
Input: root = [4, 2, 7, 1, 3, 6, 9];
Output: [4, 7, 2, 9, 6, 3, 1];
```

**Solution:**

```
function invertTree(root) {
  if (root === null) {
    return null;
  }

  const queue = [root];

  while (queue.length > 0) {
    const current = queue.shift();

    // Swap left and right children
    const temp = current.left;
    current.left = current.right;
    current.right = temp;

    // Add children to the queue
    if (current.left !== null) {
      queue.push(current.left);
    }
    if (current.right !== null) {
      queue.push(current.right);
    }
  }

  return root;
}
```

```

}

// Test
const tree = {
  val: 4,
  left: {
    val: 2,
    left: { val: 1, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
  right: {
    val: 7,
    left: { val: 6, left: null, right: null },
    right: { val: 9, left: null, right: null },
  },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{ "val":4,"left":{ "val":7,"left":{ "val":9,"left":null,"right":null },"right":{ "val":6,"left":null,"right":null } },"right":{ "val":2,"left":{ "val":3,"left":null,"right":null },"right":{ "val":1,"left":null,"right":null } }

```

### **Explanation:**

1. **Objective:** Use an iterative approach to invert the binary tree by swapping the left and right children for each node.
2. **Approach:**
  - o Use a queue to traverse the tree level by level (Breadth-First Search).
  - o For each node, swap its left and right children and add them to the queue for further processing.
3. **Steps:**
  - o Initialize a queue with the root node.
  - o While the queue is not empty:
    - Remove the first node from the queue.
    - Swap its left and right children.
    - Add its non-null children to the queue for further processing.
  - o Return the root node after all nodes are processed.
4. **Why it Works:** The queue ensures that all nodes are processed iteratively in a level-order fashion, and swapping the children at each step ensures the tree is inverted.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue, which can hold up to  $n$  nodes in the worst case (for a full binary tree).

### **Invert Binary Tree (Iterative Approach Using Stack)**

**223. Problem: Invert a binary tree using an iterative depth-first search approach with a stack.**

**Example:**

Input: root = [4, 2, 7, 1, 3, 6, 9];  
Output: [4, 7, 2, 9, 6, 3, 1];

**Solution:**

```
function invertTree(root) {  
    if (root === null) {  
        return null;  
    }  
  
    const stack = [root];  
  
    while (stack.length > 0) {  
        const current = stack.pop();  
  
        // Swap left and right children  
        const temp = current.left;  
        current.left = current.right;  
        current.right = temp;  
  
        // Add children to the stack for further processing  
        if (current.left !== null) {  
            stack.push(current.left);  
        }  
        if (current.right !== null) {  
            stack.push(current.right);  
        }  
    }  
  
    return root;  
}  
  
// Test  
const tree = {  
    val: 4,  
    left: {  
        val: 2,  
        left: { val: 1, left: null, right: null },  
        right: { val: 3, left: null, right: null },  
    },  
    right: {  
        val: 7,  
        left: { val: 6, left: null, right: null },  
        right: { val: 9, left: null, right: null },  
    },  
};
```

```

};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{"val":4,"left":{"val":7,"left":{"val":9,"left":null,"right":null},"right":{"val":6,"left":null,"right":null}),"right":{"val":2,"left":{"val":3,"left":null,"right":null},"right":{"val":1,"left":null,"right":null}}}

```

### **Explanation:**

1. **Objective:** Use a stack to traverse the binary tree iteratively in a depth-first manner and invert it by swapping left and right children.
2. **Approach:**
  - o Use a stack to store nodes to be processed.
  - o For each node, swap its left and right children.
  - o Push its non-null children onto the stack for further processing.
3. **Steps:**
  - o Initialize the stack with the root node.
  - o While the stack is not empty:
    - Pop a node from the stack.
    - Swap its left and right children.
    - Push the non-null children onto the stack.
  - o Return the root node after all nodes are processed.
4. **Why it Works:** The stack ensures that nodes are processed iteratively in a depth-first manner, and swapping the children at each step inverts the tree.
5. **Time Complexity:** O(n) — Each node is visited once.
6. **Space Complexity:** O(h) — Space for the stack, where h is the height of the tree.

### **Invert Binary Tree (Post-Order Traversal)**

**224. Problem: Invert a binary tree using post-order traversal (processing children before swapping the current node).**

#### **Solution:**

```

function invertTree(root) {
  if (root === null) {
    return null;
  }

  invertTree(root.left);
  invertTree(root.right);

  const temp = root.left;
  root.left = root.right;
  root.right = temp;

  return root;
}

```

```

// Test
const tree = {
  val: 4,
  left: {
    val: 2,
    left: { val: 1, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
  right: {
    val: 7,
    left: { val: 6, left: null, right: null },
    right: { val: 9, left: null, right: null },
  },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{"val":4,"left":{"val":7,"left":{"val":9,"left":null,"right":null},"right":{"val":6,"left":null,"right":null}),"right":{"val":2,"left":{"val":3,"left":null,"right":null},"right":{"val":1,"left":null,"right":null}}}

```

### Explanation:

1. **Objective:** Use post-order traversal to invert the binary tree by processing the left and right subtrees before swapping them at each node.
2. **Approach:**
  - o Recursively invert the left subtree.
  - o Recursively invert the right subtree.
  - o Swap the left and right children of the current node.
3. **Steps:**
  - o Base Case: If the current node is null, return null.
  - o Recursively process the left and right subtrees.
  - o Swap the left and right children for the current node.
  - o Return the root node.
4. **Why it Works:** Post-order traversal ensures that all children are inverted before swapping their positions for the current node.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Invert Binary Tree (Level-Order Traversal Using a Queue)

**225. Problem: Invert a binary tree using level-order traversal (Breadth-First Search) while swapping left and right children at each level.**

### Solution:

```

function invertTree(root) {
  if (root === null) {
    return null;
  }

  const queue = [root];

  while (queue.length > 0) {
    const current = queue.shift();

    // Swap left and right children
    const temp = current.left;
    current.left = current.right;
    current.right = temp;

    // Add children to the queue for further processing
    if (current.left !== null) {
      queue.push(current.left);
    }
    if (current.right !== null) {
      queue.push(current.right);
    }
  }

  return root;
}

// Test
const tree = {
  val: 4,
  left: {
    val: 2,
    left: { val: 1, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
  right: {
    val: 7,
    left: { val: 6, left: null, right: null },
    right: { val: 9, left: null, right: null },
  },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{ "val":4,"left":{ "val":7,"left":{ "val":9,"left":null,"right":null },"right":{ "val":6,"left":null,"right":null } }, "right":{ "val":2,"left":{ "val":3,"left":null,"right":null },"right":{ "val":1,"left":null,"right":null } } }

```

### **Explanation:**

1. **Objective:** Use a level-order traversal to process the binary tree level by level and invert it by swapping left and right children at each level.
2. **Approach:**
  - o Use a queue to perform a breadth-first traversal.
  - o For each node dequeued, swap its left and right children.
  - o Add the non-null children of the node to the queue for further processing.
3. **Steps:**
  - o Initialize the queue with the root node.
  - o While the queue is not empty:
    - Dequeue a node and swap its left and right children.
    - Add its left and right children (if not null) to the queue.
  - o Return the root node after processing all levels.
4. **Why it Works:** By processing the tree level by level and swapping children at each node, the tree is inverted in a systematic and iterative manner.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue, which can hold up to  $n$  nodes in the worst case (for a full binary tree).

### **Invert Binary Tree (In-Order Traversal)**

**226. Problem: Invert a binary tree using in-order traversal, where the left subtree is processed first, followed by the current node, and then the right subtree.**

### **Solution:**

```
function invertTree(root) {  
    if (root === null) {  
        return null;  
    }  
  
    // Process the left subtree first  
    invertTree(root.left);  
  
    // Swap left and right children  
    const temp = root.left;  
    root.left = root.right;  
    root.right = temp;  
  
    // Process the new left subtree (originally the right subtree)  
    invertTree(root.left);  
  
    return root;  
}  
  
// Test  
const tree = {  
    val: 4,  
    left: {  
        val: 2,  
        left: {  
            val: 1,  
            left: null,  
            right: null  
        },  
        right: {  
            val: 3,  
            left: null,  
            right: null  
        }  
    },  
    right: {  
        val: 5,  
        left: null,  
        right: null  
    }  
};  
invertTree(tree);  
// Output: { val: 4, left: { val: 3, left: null, right: null }, right: { val: 2, left: { val: 1, left: null, right: null }, right: null } }
```

```

    val: 2,
    left: { val: 1, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
  right: {
    val: 7,
    left: { val: 6, left: null, right: null },
    right: { val: 9, left: null, right: null },
  },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{ "val":4,"left":{ "val":7,"left":{ "val":9,"left":null,"right":null },"right":{ "val":6,"left":null,"right":null } },"right":{ "val":2,"left":{ "val":3,"left":null,"right":null },"right":{ "val":1,"left":null,"right":null } } }

```

### Explanation:

1. **Objective:** Invert the binary tree using in-order traversal by processing the left subtree first, swapping children at the current node, and then processing the new left subtree.
2. **Approach:**
  - o Traverse the left subtree recursively.
  - o Swap the left and right children at the current node.
  - o Traverse the new left subtree recursively (which was originally the right subtree before swapping).
3. **Steps:**
  - o Base Case: If the current node is null, return null.
  - o Recursively invert the left subtree.
  - o Swap the left and right children of the current node.
  - o Recursively invert the new left subtree.
  - o Return the root node.
4. **Why it Works:** In-order traversal ensures that the left subtree is fully inverted before processing the right subtree.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Invert Binary Tree (Pre-Order Traversal)

**227. Problem: Invert a binary tree using pre-order traversal, where the current node is processed first, followed by its left and right subtrees.**

```

function invertTree(root) {
  if (root === null) {
    return null;
  }
}

```

```

// Swap left and right children
const temp = root.left;
root.left = root.right;
root.right = temp;

// Recursively invert left and right subtrees
invertTree(root.left);
invertTree(root.right);

return root;
}

// Test
const tree = {
  val: 4,
  left: {
    val: 2,
    left: { val: 1, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
  right: {
    val: 7,
    left: { val: 6, left: null, right: null },
    right: { val: 9, left: null, right: null },
  },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{"val":4,"left":{"val":7,"left":{"val":9,"left":null,"right":null},"right":{"val":6,"left":null,"right":null}),"right":{"val":2,"left":{"val":3,"left":null,"right":null},"right":{"val":1,"left":null,"right":null}}}

```

### Explanation:

1. **Objective:** Invert the binary tree using pre-order traversal, processing the current node before its left and right subtrees.
2. **Approach:**
  - o Swap the left and right children of the current node immediately after visiting it.
  - o Recursively invert the left subtree.
  - o Recursively invert the right subtree.
3. **Steps:**
  - o Base Case: If the current node is null, return null.
  - o Swap the left and right children of the current node.
  - o Recursively invert the left subtree, then the right subtree.
  - o Return the root node.

4. **Why it Works:** Pre-order traversal ensures that the current node is processed before its children are inverted.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Invert Binary Tree (Post-Order Traversal Using a Stack)

**228. Problem: Invert a binary tree using an iterative post-order traversal with a stack.**

**Solution:**

```
function invertTree(root) {
  if (root === null) {
    return null;
  }

  const stack = [];
  const visited = new Set();
  stack.push(root);

  while (stack.length > 0) {
    const current = stack[stack.length - 1];

    // Post-order traversal: Process left and right subtrees before swapping
    if (current.left && !visited.has(current.left)) {
      stack.push(current.left);
    } else if (current.right && !visited.has(current.right)) {
      stack.push(current.right);
    } else {
      // Swap left and right children
      const temp = current.left;
      current.left = current.right;
      current.right = temp;

      visited.add(current);
      stack.pop();
    }
  }

  return root;
}

// Test
const tree = {
  val: 4,
  left: {
    val: 2,
    left: { val: 1, left: null, right: null },
    right: null
  }
}
```

```

        right: { val: 3, left: null, right: null },
    },
    right: {
        val: 7,
        left: { val: 6, left: null, right: null },
        right: { val: 9, left: null, right: null },
    },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{
    "val": 4,
    "left": {
        "val": 7,
        "left": {
            "val": 9,
            "left": null,
            "right": null
        },
        "right": {
            "val": 6,
            "left": null,
            "right": null
        }
    },
    "right": {
        "val": 2,
        "left": {
            "val": 3,
            "left": null,
            "right": null
        },
        "right": {
            "val": 1,
            "left": null,
            "right": null
        }
    }
}

```

### Explanation:

1. **Objective:** Use a stack to perform a post-order traversal and invert the binary tree by swapping left and right children after processing the subtrees.
2. **Approach:**
  - o Use a stack to traverse the tree iteratively.
  - o Maintain a visited set to track nodes whose children have already been processed.
  - o For each node:
    - Push its left and right children onto the stack if they haven't been processed.
    - Swap the children after processing both subtrees.
3. **Steps:**
  - o Initialize a stack with the root node.
  - o While the stack is not empty:
    - Check if the current node's children have been processed.
    - If not, push them onto the stack.
    - If processed, swap the left and right children and mark the node as visited.
  - o Return the root node after traversal.
4. **Why it Works:** The stack ensures that nodes are processed in post-order (children first, then parent), and swapping happens after processing both subtrees.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the stack, where  $h$  is the height of the tree.

### Invert Binary Tree (Using Morris Traversal for Constant Space)

**229. Problem: Invert a binary tree using Morris Traversal, which allows in-place tree traversal with  $O(1)$  space.**

### Solution:

```

function invertTree(root) {
    let current = root;

```

```

while (current !== null) {
  if (current.left !== null) {
    // Find the rightmost node in the left subtree
    let predecessor = current.left;
    while (predecessor.right !== null && predecessor.right !== current) {
      predecessor = predecessor.right;
    }

    if (predecessor.right === null) {
      // Create a temporary link to the current node
      predecessor.right = current;

      // Swap left and right children
      const temp = current.left;
      current.left = current.right;
      current.right = temp;

      // Move to the new left child
      current = current.left;
    } else {
      // Remove the temporary link
      predecessor.right = null;

      // Move to the new right child
      current = current.right;
    }
  } else {
    // If no left child, just swap and move right
    const temp = current.left;
    current.left = current.right;
    current.right = temp;

    current = current.right;
  }
}

return root;
}

// Test
const tree = {
  val: 4,
  left: {
    val: 2,
    left: { val: 1, left: null, right: null },
    right: { val: 3, left: null, right: null },
  },
}

```

```

right: {
  val: 7,
  left: { val: 6, left: null, right: null },
  right: { val: 9, left: null, right: null },
},
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{"val":4,"left":{"val":7,"left":{"val":9,"left":null,"right":null},"right":{"val":6,"left":null,"right":null}),"right":{"val":2,"left":{"val":3,"left":null,"right":null},"right":{"val":1,"left":null,"right":null}}}

```

### **Explanation:**

1. **Objective:** Invert the binary tree using Morris Traversal to achieve constant space complexity without recursion or a stack/queue.
2. **Approach:**
  - o Use the Morris Traversal technique to temporarily modify the tree structure, allowing in-place traversal.
  - o Swap the left and right children while traversing the tree.
  - o Use a temporary link to traverse back to the parent node after processing the left subtree.
3. **Steps:**
  - o For each node, check if it has a left child:
    - If yes, find its in-order predecessor (rightmost node in the left subtree).
    - Create a temporary link from the predecessor to the current node if it doesn't exist.
    - Swap the left and right children of the current node.
    - Move to the left child.
    - If the temporary link exists, remove it and move to the right child.
  - o If no left child, swap the children and move to the right child.
4. **Why it Works:** Morris Traversal modifies the tree temporarily to avoid using extra space for recursion or a stack. It restores the tree structure after traversal.
5. **Time Complexity:** O(n) — Each node is visited at most twice.
6. **Space Complexity:** O(1) — No additional space is used apart from a few variables.

### **Invert Binary Tree (Using Array for Iterative Approach)**

**230. Problem: Invert a binary tree iteratively by using an array as an alternative to a stack or queue.**

### **Solution:**

```

function invertTree(root) {
  if (root === null) {
    return null;
  }
}

```

```

const nodes = [root]; // Array to simulate stack/queue

while (nodes.length > 0) {
    const current = nodes.pop(); // Pop last element for stack-like behavior

    // Swap left and right children
    const temp = current.left;
    current.left = current.right;
    current.right = temp;

    // Add children to the array
    if (current.left !== null) {
        nodes.push(current.left);
    }
    if (current.right !== null) {
        nodes.push(current.right);
    }
}

return root;
}

// Test
const tree = {
    val: 4,
    left: {
        val: 2,
        left: { val: 1, left: null, right: null },
        right: { val: 3, left: null, right: null },
    },
    right: {
        val: 7,
        left: { val: 6, left: null, right: null },
        right: { val: 9, left: null, right: null },
    },
};

console.log(JSON.stringify(invertTree(tree)));
// Output:
{
    "val": 4,
    "left": {
        "val": 7,
        "left": {
            "val": 9,
            "left": null,
            "right": null
        },
        "right": {
            "val": 6,
            "left": null,
            "right": null
        }
    },
    "right": {
        "val": 2,
        "left": {
            "val": 3,
            "left": null,
            "right": null
        },
        "right": {
            "val": 1,
            "left": null,
            "right": null
        }
    }
}

```

### Explanation:

1. **Objective:** Use an array as a dynamic data structure to traverse and invert the binary tree iteratively.
2. **Approach:**

- Initialize an array with the root node.
  - Pop nodes from the array and swap their left and right children.
  - Add non-null children of each node to the array for further processing.
3. **Steps:**
- Start with an array containing the root node.
  - While the array is not empty:
    - Pop the last node (stack-like behavior).
    - Swap its left and right children.
    - Push non-null children to the array.
  - Return the root node after traversal.
4. **Why it Works:** Arrays can dynamically grow and shrink, allowing them to be used as an alternative to explicit stack or queue structures.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the array, which can hold up to  $n$  nodes in the worst case.

### Finding the Maximum Path in Trees (Recursive Approach)

**231. Problem: Given the root of a binary tree, find the maximum path sum.**

**A path in the binary tree is defined as any sequence of nodes starting from some node and moving downward (parent to child). The path must contain at least one node and does not need to pass through the root.**

**Example:**

```
Input: root = [-10, 9, 20, null, null, 15, 7]
Output: 42
Explanation: The path with the maximum sum is 15 → 20 → 7.
```

**Solution:**

```
function maxPathSum(root) {
  let maxSum = -Infinity;

  function findMaxPath(node) {
    if (node === null) {
      return 0;
    }

    // Recursively calculate the maximum path sum of the left and right subtrees
    const leftMax = Math.max(findMaxPath(node.left), 0); // Ignore negative paths
    const rightMax = Math.max(findMaxPath(node.right), 0); // Ignore negative paths

    // Update the global maximum path sum
    const currentPathSum = node.val + leftMax + rightMax;
    maxSum = Math.max(maxSum, currentPathSum);

    // Return the maximum path sum including the current node as part of the path
    return node.val + Math.max(leftMax, rightMax);
  }

  findMaxPath(root);
}
```

```

        return node.val + Math.max(leftMax, rightMax);
    }

    findMaxPath(root);
    return maxSum;
}

// Test
const tree = {
    val: -10,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(maxPathSum(tree)); // Output: 42

```

### Explanation:

1. **Objective:** Find the maximum path sum in the binary tree.
2. **Approach:**
  - o Use a recursive helper function to calculate the maximum path sum for each subtree.
  - o At each node:
    - Calculate the maximum path sum of the left subtree (leftMax) and right subtree (rightMax).
    - Ignore negative path sums by taking `Math.max(leftMax, 0)` and `Math.max(rightMax, 0)`.
    - Update the global maximum path sum (maxSum) using the sum of the current node value, leftMax, and rightMax.
  - o Return the maximum path sum that includes the current node and either its left or right subtree.
3. **Steps:**
  - o Base Case: If the current node is null, return 0.
  - o Recursive Case:
    - Compute leftMax and rightMax.
    - Update maxSum with the sum of the current node and both subtrees.
    - Return the maximum of the two subtrees plus the current node value.
4. **Why it Works:** This approach ensures that all paths in the tree are considered, and the global maximum path sum is updated at each step.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Finding the Maximum Path in Trees (Track Path with Maximum Sum)

**232. Problem:** In addition to finding the maximum path sum, return the nodes forming the path that produces the maximum sum.

**Example:**

```
Input: root = [-10, 9, 20, null, null, 15, 7]
Output: { maxSum: 42, path: [15, 20, 7] }
Explanation: The path with the maximum sum is 15 → 20 → 7.
```

**Solution:**

```
function maxPathSumWithPath(root) {
  let maxSum = -Infinity;
  let maxPath = [];

  function findMaxPath(node) {
    if (node === null) {
      return { sum: 0, path: [] };
    }

    // Recursively calculate the max path sum of left and right subtrees
    const left = findMaxPath(node.left);
    const right = findMaxPath(node.right);

    // Ignore negative paths
    const leftSum = Math.max(left.sum, 0);
    const rightSum = Math.max(right.sum, 0);

    // Current path sum considering this node as the root of the path
    const currentPathSum = node.val + leftSum + rightSum;

    // Update the global max sum and path
    if (currentPathSum > maxSum) {
      maxSum = currentPathSum;
      maxPath = [
        ...(leftSum > 0 ? left.path : []),
        node.val,
        ...(rightSum > 0 ? right.path : []),
      ];
    }
  }

  // Return the max sum and the path including the current node
  if (leftSum > rightSum) {
    return { sum: node.val + leftSum, path: [...left.path, node.val] };
  } else {
    return { sum: node.val + rightSum, path: [...right.path, node.val] };
  }
}
```

```

}

findMaxPath(root);
return { maxSum, path: maxPath };
}

// Test
const tree = {
  val: -10,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(maxPathSumWithPath(tree));
// Output: { maxSum: 42, path: [15, 20, 7] }

```

### Explanation:

1. **Objective:** Find the maximum path sum and track the nodes that form the path with the maximum sum.
2. **Approach:**
  - o Use a recursive function that calculates the maximum path sum and tracks the nodes in the path.
  - o At each node:
    - Compute the maximum path sum for the left and right subtrees.
    - Consider only non-negative paths ( $\text{Math.max(leftSum, 0)}$  and  $\text{Math.max(rightSum, 0)}$ ).
    - Calculate the path sum for the current node as the root of the path.
    - Update the global maximum sum and path if the current path sum exceeds the global maximum.
    - Return the maximum path sum and path that includes the current node and one of its subtrees.
3. **Steps:**
  - o Base Case: If the current node is null, return a sum of 0 and an empty path.
  - o Recursive Case:
    - Compute `left.sum`, `left.path`, `right.sum`, and `right.path`.
    - Update `maxSum` and `maxPath` using the current node and the best possible subtrees.
    - Return the sum and path for the current node including the best subtree.
4. **Why it Works:** This approach ensures that both the global maximum path sum and the corresponding path are updated as the recursion progresses.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.

6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Finding the Maximum Path in Trees (Iterative Approach)

**233. Problem: Find the maximum path sum of a binary tree using an iterative approach.**

**Example:**

Input: root = [-10, 9, 20, null, null, 15, 7]

Output: 42

Explanation: The path with the maximum sum is  $15 \rightarrow 20 \rightarrow 7$ .

**Solution:**

```
function maxPathSum(root) {
    let maxSum = -Infinity;

    const stack = [{ node: root, leftVisited: false, rightVisited: false }];
    const nodeSums = new Map();

    while (stack.length > 0) {
        const current = stack[stack.length - 1];
        const { node, leftVisited, rightVisited } = current;

        if (node === null) {
            stack.pop();
            continue;
        }

        if (!leftVisited) {
            current.leftVisited = true;
            if (node.left) {
                stack.push({
                    node: node.left,
                    leftVisited: false,
                    rightVisited: false,
                });
            }
        } else if (!rightVisited) {
            current.rightVisited = true;
            if (node.right) {
                stack.push({
                    node: node.right,
                    leftVisited: false,
                    rightVisited: false,
                });
            }
        }

        const sum = node.val + (leftVisited ? nodeSums.get(node.left) : 0) + (rightVisited ? nodeSums.get(node.right) : 0);
        nodeSums.set(node, sum);
        maxSum = Math.max(maxSum, sum);
    }

    return maxSum;
}
```

```

        }
    } else {
        stack.pop();

        const leftMax = Math.max(nodeSums.get(node.left) || 0, 0);
        const rightMax = Math.max(nodeSums.get(node.right) || 0, 0);

        const currentPathSum = node.val + leftMax + rightMax;
        maxSum = Math.max(maxSum, currentPathSum);

        nodeSums.set(node, node.val + Math.max(leftMax, rightMax));
    }

    return maxSum;
}

// Test
const tree = {
    val: -10,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(maxPathSum(tree)); // Output: 42

```

### Explanation:

1. **Objective:** Find the maximum path sum of a binary tree using an iterative depth-first approach.
2. **Approach:**
  - o Use a stack to simulate the recursive depth-first traversal of the tree.
  - o Use a nodeSums map to store the maximum path sums of visited nodes.
  - o For each node:
    - First visit its left and right subtrees using the stack.
    - Once both subtrees are visited, calculate the maximum path sum at the current node and update the global maxSum.
3. **Steps:**
  - o Initialize the stack with the root node.
  - o Use leftVisited and rightVisited flags to track whether the left and right subtrees have been visited.
  - o Calculate the path sum for each node after its children have been processed.
  - o Update maxSum to track the global maximum path sum.

4. **Why it Works:** The stack ensures that nodes are processed in post-order (left, right, then node), allowing the calculation of path sums after visiting subtrees.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the stack, where  $h$  is the height of the tree.

### Finding the Maximum Path in Trees (Single Node as Path)

**234. Problem:** Find the maximum path sum of a binary tree, but the path must consist of exactly one node.

**Example:**

```
Input: root = [-10, 9, 20, null, null, 15, 7]
Output: 20
Explanation: The maximum path sum with exactly one node is 20.
```

**Solution:**

```
function maxSingleNodePathSum(root) {
  if (root === null) {
    return -Infinity;
  }

  const leftMax = maxSingleNodePathSum(root.left);
  const rightMax = maxSingleNodePathSum(root.right);

  return Math.max(root.val, leftMax, rightMax);
}

// Test
const tree = {
  val: -10,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(maxSingleNodePathSum(tree)); // Output: 20
```

**Explanation:**

1. **Objective:** Find the maximum path sum where the path consists of exactly one node.
2. **Approach:**

- Recursively compute the maximum value among the current node and the maximum values of its left and right subtrees.
  - Return the maximum value at each step.
3. **Steps:**
- Base Case: If the current node is null, return -Infinity to ensure it doesn't contribute to the maximum path sum.
  - Recursive Case:
    - Compute the maximum values for the left and right subtrees.
    - Return the maximum of the current node value and the subtree maximums.
4. **Why it Works:** This approach ensures that only the maximum value from any subtree or the current node is considered at each step.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Finding the Maximum Path in Trees (Include and Exclude Root)

**235. Problem: Find the maximum path sum of a binary tree, considering paths that include or exclude the root.**

**Example:**

```
Input: root = [-10, 9, 20, null, null, 15, 7]
Output: 42
Explanation: The path with the maximum sum is 15 → 20 → 7.
```

**Solution:**

```
function maxPathSumIncludeExclude(root) {
  if (root === null) {
    return 0;
  }

  let maxSum = -Infinity;

  function dfs(node) {
    if (node === null) {
      return 0;
    }

    // Recursively calculate the maximum path sum for left and right subtrees
    const leftMax = Math.max(dfs(node.left), 0);
    const rightMax = Math.max(dfs(node.right), 0);

    // Include both children plus the current node
    const currentPathSum = node.val + leftMax + rightMax;

    maxSum = Math.max(maxSum, currentPathSum);
  }

  dfs(root);
  return maxSum;
}
```

```

// Update the global maximum path sum
maxSum = Math.max(maxSum, currentPathSum);

// Return the max sum including either the left or right subtree, plus the current node
return node.val + Math.max(leftMax, rightMax);
}

dfs(root);
return maxSum;
}

// Test
const tree = {
  val: -10,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(maxPathSumIncludeExclude(tree)); // Output: 42

```

### Explanation:

1. **Objective:** Calculate the maximum path sum, considering paths that may or may not include the root.
2. **Approach:**
  - o Use a depth-first search (DFS) to explore the tree.
  - o For each node:
    - Compute the maximum path sum of the left and right subtrees.
    - Ignore negative path sums by using  $\text{Math.max}(\text{dfs}(\text{node.left}), 0)$  and  $\text{Math.max}(\text{dfs}(\text{node.right}), 0)$ .
    - Update the global maximum path sum by considering the current node and both subtrees.
    - Return the maximum path sum that includes the current node and one subtree.
3. **Steps:**
  - o Base Case: If the current node is null, return 0.
  - o Recursive Case:
    - Compute leftMax and rightMax for the subtrees.
    - Update maxSum using the current node and both subtrees.
    - Return the maximum sum for the path that includes the current node and one subtree.
4. **Why it Works:** By updating the global maximum sum at each node and returning the maximum path sum for one subtree, all possible paths are considered.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.

6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Finding the Maximum Path in Trees (Exclude Negative Nodes)

**236. Problem: Modify the maximum path sum calculation to exclude any negative nodes from the path.**

**Solution:**

```
function maxPathSumExcludeNegative(root) {  
    if (root === null) {  
        return 0;  
    }  
  
    let maxSum = -Infinity;  
  
    function dfs(node) {  
        if (node === null) {  
            return 0;  
        }  
  
        // Ignore nodes with negative values  
        const leftMax = Math.max(dfs(node.left), 0);  
        const rightMax = Math.max(dfs(node.right), 0);  
  
        if (node.val < 0) {  
            return 0; // Exclude the current node if it's negative  
        }  
  
        // Calculate the path sum at the current node  
        const currentPathSum = node.val + leftMax + rightMax;  
  
        // Update the global maximum path sum  
        maxSum = Math.max(maxSum, currentPathSum);  
  
        // Return the maximum sum including one subtree and the current node  
        return node.val + Math.max(leftMax, rightMax);  
    }  
  
    dfs(root);  
    return maxSum;  
}  
  
// Test  
const tree = {  
    val: -10,  
    left: { val: 9, left: null, right: null },  
    right: {  
        val: 10,  
        left: { val: 8, left: null, right: null },  
        right: {  
            val: 11,  
            left: null,  
            right: null  
        }  
    }  
};  
maxPathSumExcludeNegative(tree);
```

```

val: 20,
left: { val: -15, left: null, right: null },
right: { val: 7, left: null, right: null },
},
};

console.log(maxPathSumExcludeNegative(tree)); // Output: 36

```

### Explanation:

1. **Objective:** Exclude nodes with negative values from contributing to the path sum.
2. **Approach:**
  - o Use DFS to calculate the maximum path sum while ignoring negative nodes.
  - o If a node value is negative, return 0 for that path.
3. **Steps:**
  - o Base Case: If the node is null, return 0.
  - o Recursive Case:
    - Compute the maximum path sum for the left and right subtrees, excluding negative nodes.
    - If the current node's value is negative, return 0 (exclude it from the path).
    - Update the global maximum sum by considering the current node and both subtrees.
    - Return the maximum path sum for the current node and one subtree.
4. **Why it Works:** This ensures that only positive values contribute to the path sum, avoiding paths that would decrease the sum.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Finding the Maximum Path in Trees (With Node Count Constraint)

**237. Problem:** **Find the maximum path sum of a binary tree, but the path must consist of at least k nodes.**

### Example:

```

Input: root = [-10, 9, 20, null, null, 15, 7], k = 2
Output: 42
Explanation: The path 15 → 20 → 7 satisfies the condition and produces the maximum sum.

```

### Solution:

```

function maxPathSumWithNodeConstraint(root, k) {
  let maxSum = -Infinity;

  function dfs(node) {

```

```

if (node === null) {
    return { maxSum: 0, nodeCount: 0 };
}

// Get left and right subtree results
const left = dfs(node.left);
const right = dfs(node.right);

// Calculate current path sum and node count
const currentSum = node.val + left.maxSum + right.maxSum;
const currentNodeCount = 1 + left.nodeCount + right.nodeCount;

// Update the global maxSum only if the node count >= k
if (currentNodeCount >= k) {
    maxSum = Math.max(maxSum, currentSum);
}

// Return the best path sum and node count for the current node
const bestSinglePathSum = node.val + Math.max(left.maxSum, right.maxSum);
const bestNodeCount = 1 + Math.max(left.nodeCount, right.nodeCount);

return {
    maxSum: Math.max(bestSinglePathSum, 0), // Ignore negative sums
    nodeCount: bestNodeCount,
};

dfs(root);
return maxSum;
}

// Test
const tree = {
    val: -10,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(maxPathSumWithNodeConstraint(tree, 2)); // Output: 42

```

### Explanation:

1. **Objective:** Calculate the maximum path sum, ensuring that the path contains at least  $k$  nodes.

## 2. Approach:

- o Modify the DFS function to return both the maximum path sum and the node count for each subtree.
- o Use the node count to ensure the global maximum sum is updated only when the path contains at least k nodes.
- o Return the best single path sum and its corresponding node count at each step.

## 3. Steps:

- o Base Case: If the current node is null, return { maxSum: 0, nodeCount: 0 }.
- o Recursive Case:
  - Compute the maximum path sum and node count for the left and right subtrees.
  - Calculate the current path sum and node count, including the current node and both subtrees.
  - Update the global maximum sum if the node count is at least k.
  - Return the best single path sum and node count for the current node.

4. Why it Works: By tracking the node count along with the path sum, this approach ensures that only valid paths (with at least k nodes) are considered for the global maximum.

5. Time Complexity: O(n) — Each node is visited once.

6. Space Complexity: O(h) — Space for the recursion stack, where h is the height of the tree.

## Finding the Maximum Path in Trees (Longest Path Only)

**238. Problem:** **Find the maximum path sum in a binary tree where the path must include the largest number of nodes possible.**

### Example:

Input: root = [-10, 9, 20, null, null, 15, 7]

Output: 35

Explanation: The path with the most nodes is 15 → 20 → 7 → -10.

### Solution:

```
function maxPathSumLongestPath(root) {  
    let maxSum = -Infinity;  
    let maxNodeCount = 0;  
  
    function dfs(node) {  
        if (node === null) {  
            return { maxSum: 0, nodeCount: 0 };  
        }  
  
        // Get left and right subtree results  
        const left = dfs(node.left);  
        const right = dfs(node.right);  
    }  
}
```

```

// Calculate current path sum and node count
const currentSum = node.val + left.maxSum + right.maxSum;
const currentNodeCount = 1 + left.nodeCount + right.nodeCount;

// Update maxSum if this path has the largest node count or equal node count with higher
sum
if (
  currentNodeCount > maxNodeCount ||
  (currentNodeCount === maxNodeCount && currentSum > maxSum)
) {
  maxSum = currentSum;
  maxNodeCount = currentNodeCount;
}

// Return the best single path sum and its node count
const bestSinglePathSum = node.val + Math.max(left.maxSum, right.maxSum);
const bestNodeCount = 1 + Math.max(left.nodeCount, right.nodeCount);

return {
  maxSum: Math.max(bestSinglePathSum, 0), // Ignore negative sums
  nodeCount: bestNodeCount,
};

dfs(root);
return maxSum;
}

// Test
const tree = {
  val: -10,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(maxPathSumLongestPath(tree)); // Output: 35

```

### Explanation:

1. **Objective:** Calculate the maximum path sum for the path that includes the most nodes in the binary tree.
2. **Approach:**
  - o Use DFS to track both the maximum path sum and the node count for each subtree.

- Update the global maximum path sum only if the current path has more nodes or an equal number of nodes with a higher sum.
3. **Steps:**
- Base Case: If the current node is null, return { maxSum: 0, nodeCount: 0 }.
  - Recursive Case:
    - Compute the maximum path sum and node count for the left and right subtrees.
    - Calculate the path sum and node count, including the current node and both subtrees.
    - Update the global maximum sum and node count based on the current path.
  - Return the best single path sum and its node count for the current node.
4. **Why it Works:** This approach ensures that paths with the largest number of nodes are prioritized, and ties are resolved based on the path sum.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### **Finding the Maximum Path in Trees (Path Between Any Two Nodes)**

**239. Problem: Find the maximum path sum in a binary tree where the path can start and end at any two nodes (not necessarily passing through the root).**

**Example:**

```
Input: root = [-10, 9, 20, null, null, 15, 7]
Output: 42
Explanation: The path with the maximum sum is 15 → 20 → 7.
```

**Solution:**

```
function maxPathSumAnyTwoNodes(root) {
  let maxSum = -Infinity;

  function dfs(node) {
    if (node === null) {
      return 0;
    }

    // Recursively calculate the maximum path sums of the left and right subtrees
    const leftMax = Math.max(dfs(node.left), 0); // Ignore negative paths
    const rightMax = Math.max(dfs(node.right), 0); // Ignore negative paths

    // Calculate the current path sum including the current node
    const currentPathSum = node.val + leftMax + rightMax;

    // Update the global maximum path sum
    maxSum = Math.max(maxSum, currentPathSum);
  }

  dfs(root);
  return maxSum;
}
```

```

// Return the maximum path sum including the current node and one subtree
return node.val + Math.max(leftMax, rightMax);
}

dfs(root);
return maxSum;
}

// Test
const tree = {
  val: -10,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(maxPathSumAnyTwoNodes(tree)); // Output: 42

```

### Explanation:

1. **Objective:** Find the maximum path sum in a binary tree where the path can start and end at any two nodes.
2. **Approach:**
  - o Use a depth-first search (DFS) to explore the tree.
  - o At each node:
    - Compute the maximum path sum for the left and right subtrees.
    - Ignore negative path sums by taking the maximum of 0 and the path sum.
    - Update the global maximum path sum using the current node and both subtrees.
    - Return the maximum path sum that includes the current node and one subtree.
3. **Steps:**
  - o Base Case: If the current node is null, return 0.
  - o Recursive Case:
    - Compute the maximum path sums for the left and right subtrees.
    - Calculate the current path sum, including the current node and both subtrees.
    - Update the global maximum path sum.
    - Return the maximum path sum that includes the current node and one subtree.
4. **Why it Works:** This approach considers all possible paths in the tree and updates the global maximum path sum at each node.
5. **Time Complexity:** O(n) — Each node is visited once.

6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Finding the Maximum Path in Trees (Nodes at Even Depth)

**240. Problem:** **Find the maximum path sum in a binary tree where the path can only include nodes at even depths.**

**Example:**

Input: root = [10, 2, 10, null, null, 1, -25, null, null, null, null, 3, 4]

Output: 22

Explanation: The path with the maximum sum is  $10 \rightarrow 10 \rightarrow 2$ .

**Solution:**

```
function maxPathSumEvenDepth(root) {  
    let maxSum = -Infinity;  
  
    function dfs(node, depth) {  
        if (node === null) {  
            return 0;  
        }  
  
        // Only consider nodes at even depth  
        const includeCurrent = depth % 2 === 0 ? node.val : 0;  
  
        // Recursively calculate the maximum path sum for left and right subtrees  
        const leftMax = Math.max(dfs(node.left, depth + 1), 0);  
        const rightMax = Math.max(dfs(node.right, depth + 1), 0);  
  
        // Calculate the current path sum including the current node  
        const currentPathSum = includeCurrent + leftMax + rightMax;  
  
        // Update the global maximum path sum  
        maxSum = Math.max(maxSum, currentPathSum);  
  
        // Return the maximum path sum including the current node and one subtree  
        return includeCurrent + Math.max(leftMax, rightMax);  
    }  
  
    dfs(root, 0);  
    return maxSum;  
}  
  
// Test  
const tree = {  
    val: 10,  
}
```

```

left: { val: 2, left: null, right: null },
right: {
  val: 10,
  left: { val: 1, left: null, right: null },
  right: {
    val: -25,
    left: { val: 3, left: null, right: null },
    right: { val: 4, left: null, right: null },
  },
},
};

console.log(maxPathSumEvenDepth(tree)); // Output: 22

```

### Explanation:

1. **Objective:** Find the maximum path sum in a binary tree where the path can only include nodes at even depths.
2. **Approach:**
  - o Use DFS to explore the tree while tracking the depth of each node.
  - o Only include the current node value in the path sum if the depth is even.
  - o Compute the maximum path sum for the left and right subtrees.
  - o Update the global maximum path sum using the current node and both subtrees.
3. **Steps:**
  - o Base Case: If the current node is null, return 0.
  - o Recursive Case:
    - Check if the current depth is even. If so, include the node value in the path sum.
    - Compute the maximum path sums for the left and right subtrees.
    - Update the global maximum path sum using the current node and both subtrees.
    - Return the maximum path sum that includes the current node and one subtree.
4. **Why it Works:** By explicitly checking the depth at each node, this approach ensures that only nodes at even depths are included in the path sum.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(h)$  — Space for the recursion stack, where  $h$  is the height of the tree.

### Level-Wise Traversal of Trees (Using Queue)

**241. Problem:** Given the root of a binary tree, return its level order traversal as a list of lists, where each list represents the nodes at one level.

### Example:

Input: root = [3, 9, 20, null, null, 15, 7]

Output: [[3], [9, 20], [15, 7]]

### Solution:

```
function levelOrder(root) {  
    if (root === null) {  
        return [];  
    }  
  
    const result = [];  
    const queue = [root];  
  
    while (queue.length > 0) {  
        const levelSize = queue.length;  
        const currentLevel = [];  
  
        for (let i = 0; i < levelSize; i++) {  
            const node = queue.shift();  
            currentLevel.push(node.val);  
  
            if (node.left) {  
                queue.push(node.left);  
            }  
  
            if (node.right) {  
                queue.push(node.right);  
            }  
        }  
  
        result.push(currentLevel);  
    }  
  
    return result;  
}  
  
// Test  
const tree = {  
    val: 3,  
    left: { val: 9, left: null, right: null },  
    right: {  
        val: 20,  
        left: { val: 15, left: null, right: null },  
        right: { val: 7, left: null, right: null },  
    },  
};  
  
console.log(levelOrder(tree)); // Output: [[3], [9, 20], [15, 7]]
```

## **Explanation:**

1. **Objective:** Traverse the binary tree level by level and return the values as a list of lists.
2. **Approach:**
  - o Use a queue to process nodes level by level.
  - o For each level:
    - Count the number of nodes at the current level (levelSize).
    - Process each node, adding its value to a temporary array and enqueueing its children for the next level.
  - o Append the temporary array to the result list after processing each level.
3. **Steps:**
  - o Initialize the result list and the queue with the root node.
  - o While the queue is not empty:
    - Determine the number of nodes at the current level.
    - Iterate through the nodes, adding their values to the current level array and enqueueing their children.
  - o Return the result list after all levels are processed.
4. **Why it Works:** The queue ensures nodes are processed in the order they appear at each level, and the loop captures the values of nodes at the same level.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

## **Level-Wise Traversal of Trees (Reverse Level Order)**

**242. Problem:** Given the root of a binary tree, return its reverse level order traversal as a list of lists, where each list represents the nodes at one level starting from the bottom level.

### **Example:**

Input: root = [3, 9, 20, null, null, 15, 7];  
Output: [[15, 7], [9, 20], [3]];

### **Solution:**

```
function reverseLevelOrder(root) {  
    if (root === null) {  
        return [];  
    }  
  
    const result = [];  
    const queue = [root];  
  
    while (queue.length > 0) {  
        const levelSize = queue.length;
```

```

const currentLevel = [];

for (let i = 0; i < levelSize; i++) {
  const node = queue.shift();
  currentLevel.push(node.val);

  if (node.left) {
    queue.push(node.left);
  }
  if (node.right) {
    queue.push(node.right);
  }
}

result.unshift(currentLevel); // Insert at the beginning to reverse the order
}

return result;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(reverseLevelOrder(tree)); // Output: [[15, 7], [9, 20], [3]]

```

### Explanation:

1. **Objective:** Traverse the binary tree in reverse level order, from the bottom level to the top.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o For each level, collect node values in a temporary array.
  - o Instead of appending this array to the result, insert it at the beginning.
3. **Steps:**
  - o Initialize the result array and queue with the root node.
  - o While the queue is not empty:
    - Determine the number of nodes at the current level (levelSize).
    - Collect values from nodes at the current level.
    - Insert the collected values at the beginning of the result array.
  - o Return the result after processing all levels.

4. **Why it Works:** By inserting each level at the beginning of the result list, the traversal order is effectively reversed.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Level-Wise Traversal of Trees (Zigzag Level Order)

**243. Problem:** Given the root of a binary tree, return its zigzag level order traversal as a list of lists. The nodes at each level alternate between left-to-right and right-to-left order.

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7];
Output: [[3], [20, 9], [15, 7]];
```

**Solution:**

```
function zigzagLevelOrder(root) {
  if (root === null) {
    return [];
  }

  const result = [];
  const queue = [root];
  let leftToRight = true;

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();

      if (leftToRight) {
        currentLevel.push(node.val);
      } else {
        currentLevel.unshift(node.val);
      }

      if (node.left) {
        queue.push(node.left);
      }
      if (node.right) {
        queue.push(node.right);
      }
    }

    result.push(currentLevel);
    leftToRight = !leftToRight;
  }

  return result;
}
```

```

        result.push(currentLevel);
        leftToRight = !leftToRight; // Toggle direction
    }

    return result;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(zigzagLevelOrder(tree)); // Output: [[3], [20, 9], [15, 7]]

```

### **Explanation:**

1. **Objective:** Traverse the binary tree in a zigzag level order, alternating the direction at each level.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o Use a boolean flag (leftToRight) to track the direction of traversal at each level.
  - o Toggle the direction after processing each level.
3. **Steps:**
  - o Initialize the result array and queue with the root node.
  - o While the queue is not empty:
    - Determine the number of nodes at the current level (levelSize).
    - Collect node values in the order specified by leftToRight.
    - Append the collected values to the result array.
    - Toggle leftToRight for the next level.
  - o Return the result after processing all levels.
4. **Why it Works:** Alternating the insertion order at each level creates the zigzag pattern in the traversal.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### **Level-Wise Traversal of Trees (With Depth Information)**

**244. Problem:** **Return the level order traversal of a binary tree, but include depth information for each level.**

### **Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7];
```

```
Output: [
```

```
  [{ value: 3, depth: 0 }],
  [
    { value: 9, depth: 1 },
    { value: 20, depth: 1 },
  ],
  [
    { value: 15, depth: 2 },
    { value: 7, depth: 2 },
  ],
];

```

### Solution:

```
function levelOrderWithDepth(root) {
  if (root === null) {
    return [];
  }

  const result = [];
  const queue = [{ node: root, depth: 0 }];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const { node, depth } = queue.shift();
      currentLevel.push({ value: node.val, depth });

      if (node.left) {
        queue.push({ node: node.left, depth: depth + 1 });
      }
      if (node.right) {
        queue.push({ node: node.right, depth: depth + 1 });
      }
    }

    result.push(currentLevel);
  }

  return result;
}

// Test
const tree = {
```

```

val: 3,
left: { val: 9, left: null, right: null },
right: {
  val: 20,
  left: { val: 15, left: null, right: null },
  right: { val: 7, left: null, right: null },
},
};

console.log(levelOrderWithDepth(tree));
// Output:
// [
//   [{ value: 3, depth: 0 }],
//   [{ value: 9, depth: 1 }, { value: 20, depth: 1 }],
//   [{ value: 15, depth: 2 }, { value: 7, depth: 2 }]
// ]

```

### Explanation:

1. **Objective:** Perform level order traversal and include depth information for each node.
2. **Approach:**
  - o Use a queue to process nodes level by level.
  - o Each node in the queue is represented as an object with node and depth properties.
  - o For each node processed, add its value and depth to the current level array.
  - o Add its children to the queue with an incremented depth.
3. **Steps:**
  - o Initialize the result array and the queue with the root node and a depth of 0.
  - o While the queue is not empty:
    - Process all nodes at the current level.
    - For each node, store its value and depth, and enqueue its children with the updated depth.
  - o Append the processed level to the result array.
4. **Why it Works:** Tracking depth alongside nodes ensures that depth information is included for each node in the traversal.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Level-Wise Traversal of Trees (Average of Levels)

**245. Problem:** **Return the average of the values of nodes at each level in a binary tree.**

### Example:

Input: root = [3, 9, 20, null, null, 15, 7]

Output: [3, 14.5, 11]

Explanation:

- Level 0: [3], average = 3

```
- Level 1: [9, 20], average = (9 + 20) / 2 = 14.5
- Level 2: [15, 7], average = (15 + 7) / 2 = 11
```

### Solution:

```
function averageOfLevels(root) {
    if (root === null) {
        return [];
    }

    const averages = [];
    const queue = [root];

    while (queue.length > 0) {
        const levelSize = queue.length;
        let levelSum = 0;

        for (let i = 0; i < levelSize; i++) {
            const node = queue.shift();
            levelSum += node.val;

            if (node.left) {
                queue.push(node.left);
            }
            if (node.right) {
                queue.push(node.right);
            }
        }

        averages.push(levelSum / levelSize);
    }

    return averages;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(averageOfLevels(tree)); // Output: [3, 14.5, 11]
```

## Solution:

### Explanation:

1. **Objective:** Calculate the average of the values of nodes at each level in the binary tree.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o For each level, compute the sum of node values and divide by the number of nodes at that level.
3. **Steps:**
  - o Initialize the averages array and the queue with the root node.
  - o While the queue is not empty:
    - Calculate the number of nodes at the current level (levelSize).
    - Compute the sum of node values at the current level.
    - Divide the sum by the number of nodes to compute the average.
    - Append the average to the averages array.
  - o Return the averages array.
4. **Why it Works:** By calculating the sum and count of nodes at each level, this approach directly computes the average for each level.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

## Level-Wise Traversal of Trees (Nodes at Odd Levels)

**246. Problem:** **Return the level order traversal of a binary tree, but only include the nodes at odd levels (starting from level 1 as odd).**

### Example:

```
Input: root = [3, 9, 20, null, null, 15, 7]
Output: [[9, 20]]
Explanation: Nodes at level 1 (zero-indexed) are [9, 20].
```

## Solution:

```
function oddLevelOrder(root) {
  if (root === null) {
    return [];
  }

  const result = [];
  const queue = [root];
  let level = 0;
```

```

while (queue.length > 0) {
  const levelSize = queue.length;
  const currentLevel = [];

  for (let i = 0; i < levelSize; i++) {
    const node = queue.shift();
    if (level % 2 === 1) {
      currentLevel.push(node.val);
    }

    if (node.left) {
      queue.push(node.left);
    }
    if (node.right) {
      queue.push(node.right);
    }
  }

  if (currentLevel.length > 0) {
    result.push(currentLevel);
  }

  level++;
}

return result;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(oddLevelOrder(tree)); // Output: [[9, 20]]

```

### Explanation:

1. **Objective:** Extract nodes at odd levels (starting from level 1 as odd) during level order traversal.
2. **Approach:**
  - o Use a queue for level order traversal.
  - o Maintain a level counter to identify odd levels.

- Collect values for odd levels only and add them to the result list.
3. **Steps:**
- Initialize the result array and the queue with the root node.
  - Use a loop to process nodes level by level.
  - For each level:
    - If the level is odd, collect the node values in currentLevel.
    - Add children of nodes to the queue.
  - Increment the level counter after processing each level.
  - Return the result after processing all levels.
4. **Why it Works:** By checking the parity of the level counter, nodes at odd levels are selectively included in the result.
5. **Time Complexity:** O(n) — Each node is visited once.
6. **Space Complexity:** O(n) — Space for the queue and result list.

### Level-Wise Traversal of Trees (Nodes at Even Levels)

**247. Problem: Return the level order traversal of a binary tree, but only include the nodes at even levels (starting from level 0 as even).**

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7]
Output: [[3], [15, 7]]
Explanation: Nodes at level 0 and level 2 are included in the result.
```

**Solution:**

```
function evenLevelOrder(root) {
  if (root === null) {
    return [];
  }

  const result = [];
  const queue = [root];
  let level = 0;

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      if (level % 2 === 0) {
        currentLevel.push(node.val);
      }

      if (node.left) {
        queue.push(node.left);
      }
    }

    result.push(currentLevel);
    level++;
  }

  return result;
}
```

```

    }
    if (node.right) {
      queue.push(node.right);
    }
  }

  if (currentLevel.length > 0) {
    result.push(currentLevel);
  }

  level++;
}

return result;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(evenLevelOrder(tree)); // Output: [[3], [15, 7]]

```

### Explanation:

1. **Objective:** Extract nodes at even levels (starting from level 0 as even) during level order traversal.
2. **Approach:**
  - o Use a queue for level order traversal.
  - o Maintain a level counter to identify even levels.
  - o Collect values for even levels only and add them to the result list.
3. **Steps:**
  - o Initialize the result array and the queue with the root node.
  - o Use a loop to process nodes level by level.
  - o For each level:
    - If the level is even, collect the node values in currentLevel.
    - Add children of nodes to the queue.
  - o Increment the level counter after processing each level.
  - o Return the result after processing all levels.
4. **Why it Works:** By checking the parity of the level counter, nodes at even levels are selectively included in the result.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.

6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Level-Wise Traversal of Trees (Flattened Into a Single List)

**248. Problem:** Given the root of a binary tree, return its level order traversal flattened into a single list.

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7]
Output: [3, 9, 20, 15, 7]
```

**Solution:**

```
function levelOrderFlattened(root) {
    if (root === null) {
        return [];
    }

    const result = [];
    const queue = [root];

    while (queue.length > 0) {
        const node = queue.shift();
        result.push(node.val);

        if (node.left) {
            queue.push(node.left);
        }
        if (node.right) {
            queue.push(node.right);
        }
    }

    return result;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};
```

```
console.log(levelOrderFlattened(tree)); // Output: [3, 9, 20, 15, 7]
```

### Explanation:

1. **Objective:** Perform level order traversal and return a flattened list of node values instead of separating by levels.
2. **Approach:**
  - o Use a queue to traverse the tree level by level.
  - o For each node, append its value directly to the result list.
  - o Enqueue the left and right children of the node for further processing.
3. **Steps:**
  - o Initialize the result list and the queue with the root node.
  - o While the queue is not empty:
    - Dequeue a node, append its value to result, and enqueue its children.
  - o Return the result list.
4. **Why it Works:** The queue ensures nodes are processed in level order, and the direct appending to result creates a flattened list.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Level-Wise Traversal of Trees (Maximum Value Per Level)

**249. Problem:** Given the root of a binary tree, return the maximum value of nodes at each level.

### Example:

Input: root = [3, 9, 20, null, null, 15, 7]

Output: [3, 20, 15]

Explanation:

- Level 0: [3], max = 3
- Level 1: [9, 20], max = 20
- Level 2: [15, 7], max = 15

### Solution:

```
function maxPerLevel(root) {  
    if (root === null) {  
        return [];  
    }  
  
    const maxValues = [];  
    const queue = [root];  
  
    while (queue.length > 0) {  
        const levelSize = queue.length;
```

```

let maxValue = -Infinity;

for (let i = 0; i < levelSize; i++) {
    const node = queue.shift();
    maxValue = Math.max(maxValue, node.val);

    if (node.left) {
        queue.push(node.left);
    }
    if (node.right) {
        queue.push(node.right);
    }
}

maxValues.push(maxValue);
}

return maxValues;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(maxPerLevel(tree)); // Output: [3, 20, 15]

```

### Explanation:

1. **Objective:** Find the maximum value of nodes at each level during level order traversal.
2. **Approach:**
  - o Use a queue for level order traversal.
  - o For each level:
    - Initialize maxValue as -Infinity.
    - Process all nodes at the current level, updating maxValue for each node.
    - Append the maxValue for the level to the result list.
3. **Steps:**
  - o Initialize the maxValues list and the queue with the root node.
  - o For each level:
    - Traverse all nodes, updating the maxValue.

- Append the maximum value for the level to the result list.
  - Return the maxValues list.
- Why it Works:** Tracking the maximum value during level traversal ensures that the largest value at each level is captured.
  - Time Complexity:** O(n) — Each node is visited once.
  - Space Complexity:** O(n) — Space for the queue and result list.

### Level-Wise Traversal of Trees (Sum of Levels)

**250. Problem:** Given the root of a binary tree, return the sum of the values of nodes at each level.

**Example:**

Input: root = [3, 9, 20, null, null, 15, 7]

Output: [3, 29, 22]

Explanation:

- Level 0: [3], sum = 3
- Level 1: [9, 20], sum = 29
- Level 2: [15, 7], sum = 22

**Solution:**

```
function sumPerLevel(root) {
  if (root === null) {
    return [];
  }

  const sums = [];
  const queue = [root];

  while (queue.length > 0) {
    const levelSize = queue.length;
    let levelSum = 0;

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      levelSum += node.val;

      if (node.left) {
        queue.push(node.left);
      }
      if (node.right) {
        queue.push(node.right);
      }
    }

    sums.push(levelSum);
  }

  return sums;
}
```

```

    }

    return sums;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(sumPerLevel(tree)); // Output: [3, 29, 22]

```

### **Explanation:**

1. **Objective:** Compute the sum of node values at each level during level order traversal.
2. **Approach:**
  - o Use a queue for level order traversal.
  - o For each level:
    - Initialize levelSum as 0.
    - Add the value of each node at the current level to levelSum.
    - Append the levelSum to the result list.
3. **Steps:**
  - o Initialize the sums list and the queue with the root node.
  - o Traverse each level, summing the node values and appending the sum to sums.
  - o Return the sums list.
4. **Why it Works:** Summing values during level traversal ensures that the total value for each level is captured.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### **Level-Wise Traversal of Trees (Number of Nodes Per Level)**

#### **251. Problem: Given the root of a binary tree, return the number of nodes at each level.**

##### **Example:**

Input: root = [3, 9, 20, null, null, 15, 7]

Output: [1, 2, 2]

Explanation:

- Level 0: [3], number of nodes = 1
- Level 1: [9, 20], number of nodes = 2
- Level 2: [15, 7], number of nodes = 2

### Solution:

```
function nodesPerLevel(root) {  
    if (root === null) {  
        return [];  
    }  
  
    const nodeCounts = [];  
    const queue = [root];  
  
    while (queue.length > 0) {  
        const levelSize = queue.length;  
  
        // Add the number of nodes at this level  
        nodeCounts.push(levelSize);  
  
        for (let i = 0; i < levelSize; i++) {  
            const node = queue.shift();  
  
            if (node.left) {  
                queue.push(node.left);  
            }  
            if (node.right) {  
                queue.push(node.right);  
            }  
        }  
  
        return nodeCounts;  
    }  
  
// Test  
const tree = {  
    val: 3,  
    left: { val: 9, left: null, right: null },  
    right: {  
        val: 20,  
        left: { val: 15, left: null, right: null },  
        right: { val: 7, left: null, right: null },  
    },  
};  
  
console.log(nodesPerLevel(tree)); // Output: [1, 2, 2]
```

### Explanation:

1. **Objective:** Compute the number of nodes at each level of a binary tree.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o For each level, determine the size of the queue (levelSize), which corresponds to the number of nodes at that level.
  - o Append the size to the nodeCounts array.
3. **Steps:**
  - o Initialize the nodeCounts array and the queue with the root node.
  - o While the queue is not empty:
    - Record the number of nodes at the current level (levelSize).
    - Process each node at the current level, adding its children to the queue.
  - o Return the nodeCounts array after processing all levels.
4. **Why it Works:** The size of the queue at the start of each level traversal represents the number of nodes at that level.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### **Level-Wise Traversal of Trees (List of Leaf Nodes Per Level)**

**252. Problem: Return a list of all leaf nodes grouped by their levels in a binary tree.**

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7]
Output: [[9], [15, 7]]
Explanation:
- Level 0: No leaf nodes.
- Level 1: [9] is a leaf node.
- Level 2: [15, 7] are leaf nodes.
```

**Solution:**

```
function leafNodesPerLevel(root) {
  if (root === null) {
    return [];
  }

  const leaves = [];
  const queue = [root];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevelLeaves = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();

      // Check if the node is a leaf
```

```

if (!node.left && !node.right) {
    currentLevelLeaves.push(node.val);
}

if (node.left) {
    queue.push(node.left);
}
if (node.right) {
    queue.push(node.right);
}

if (currentLevelLeaves.length > 0) {
    leaves.push(currentLevelLeaves);
}

return leaves;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(leafNodesPerLevel(tree)); // Output: [[9], [15, 7]]

```

### Explanation:

1. **Objective:** Collect all leaf nodes in a binary tree grouped by their levels.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o For each level, identify nodes without children (leaf nodes) and add them to a temporary list.
  - o Append the temporary list to the result list after processing each level if it contains any leaf nodes.
3. **Steps:**
  - o Initialize the leaves array and the queue with the root node.
  - o While the queue is not empty:
    - Process nodes at the current level, identifying leaf nodes and adding them to currentLevelLeaves.
    - Add non-leaf children to the queue for the next level.

- Return the leaves array after processing all levels.
4. **Why it Works:** By checking the absence of children for each node, this approach identifies and groups leaf nodes at their respective levels.
  5. **Time Complexity:**  $O(n)$  — Each node is visited once.
  6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Level-Wise Traversal of Trees (Paths for Each Level)

**253. Problem:** **Return a list of paths for each level in a binary tree. Each path represents the nodes from root to the current level.**

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7]
Output: [[3], [3, 9, 20], [3, 9, 15, 3, 9, 7]]
Explanation:
- Level 0: Path is [3].
- Level 1: Paths are [3, 9] and [3, 20].
- Level 2: Paths are [3, 9, 15] and [3, 9, 7].
```

**Solution:**

```
function levelOrderPaths(root) {
  if (root === null) {
    return [];
  }

  const result = [];
  const queue = [{ node: root, path: [root.val] }];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const { node, path } = queue.shift();
      currentLevel.push(path);

      if (node.left) {
        queue.push({ node: node.left, path: [...path, node.left.val] });
      }
      if (node.right) {
        queue.push({ node: node.right, path: [...path, node.right.val] });
      }
    }

    result.push(currentLevel);
  }
}
```

```

    return result;
}

// Test
const tree = {
  val: 3,
  left: { val: 9, left: null, right: null },
  right: {
    val: 20,
    left: { val: 15, left: null, right: null },
    right: { val: 7, left: null, right: null },
  },
};

console.log(levelOrderPaths(tree));
// Output: [[[3]], [[3, 9], [3, 20]], [[3, 20, 15], [3, 20, 7]]]

```

### **Explanation:**

1. **Objective:** Return all paths from the root to nodes at each level.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o Track the path for each node as an array from the root to the current node.
  - o For each node, append its path to the current level list and add its children to the queue with updated paths.
3. **Steps:**
  - o Initialize the result array and the queue with the root node and its path.
  - o For each level:
    - Process nodes and track their paths.
    - Append the paths to the current level list.
    - Add children to the queue with their updated paths.
  - o Return the result array after processing all levels.
4. **Why it Works:** By maintaining a path for each node, this approach captures all paths from the root to the current level.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### **Level-Wise Traversal of Trees (Maximum Depth)**

#### **254. Problem: Return the maximum depth of the binary tree using level order traversal.**

##### **Example:**

Input: root = [3, 9, 20, null, null, 15, 7]  
Output: 3  
Explanation: The tree has three levels, so the maximum depth is 3.

### Solution:

```
function maxDepthLevelOrder(root) {  
    if (root === null) {  
        return 0;  
    }  
  
    const queue = [root];  
    let depth = 0;  
  
    while (queue.length > 0) {  
        const levelSize = queue.length;  
        depth++;  
  
        for (let i = 0; i < levelSize; i++) {  
            const node = queue.shift();  
  
            if (node.left) {  
                queue.push(node.left);  
            }  
            if (node.right) {  
                queue.push(node.right);  
            }  
        }  
  
        return depth;  
    }  
  
// Test  
const tree = {  
    val: 3,  
    left: { val: 9, left: null, right: null },  
    right: {  
        val: 20,  
        left: { val: 15, left: null, right: null },  
        right: { val: 7, left: null, right: null },  
    },  
};  
  
console.log(maxDepthLevelOrder(tree)); // Output: 3
```

### Explanation:

1. **Objective:** Compute the maximum depth of a binary tree using level order traversal.
2. **Approach:**

- Use a queue to process nodes level by level.
  - Increment the depth counter for each level.
  - Process all nodes at the current level, adding their children to the queue for the next level.
3. **Steps:**
- Initialize depth to 0 and the queue with the root node.
  - While the queue is not empty:
    - Increment the depth counter for each level.
    - Process all nodes at the current level.
  - Return the depth after processing all levels.
4. **Why it Works:** By counting the levels during traversal, this approach directly computes the depth of the tree.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue.

### **Level-Wise Traversal of Trees (Nodes at a Specific Depth)**

**255. Problem:** Given the root of a binary tree and a specific depth  $d$ , return all the nodes at depth  $d$ .

**Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7], d = 2
Output: [15, 7]
Explanation: Nodes at depth 2 (zero-indexed) are [15, 7].
```

**Solution:**

```
function nodesAtDepth(root, d) {
  if (root === null) {
    return [];
  }

  const queue = [{ node: root, depth: 0 }];
  const result = [];

  while (queue.length > 0) {
    const { node, depth } = queue.shift();

    if (depth === d) {
      result.push(node.val);
    }

    if (node.left) {
      queue.push({ node: node.left, depth: depth + 1 });
    }
    if (node.right) {
      queue.push({ node: node.right, depth: depth + 1 });
    }
  }

  return result;
}
```

```

        }
    }

    return result;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(nodesAtDepth(tree, 2)); // Output: [15, 7]

```

### Explanation:

1. **Objective:** Extract nodes at a specific depth  $d$  in a binary tree.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o Track the depth of each node using a depth property in the queue.
  - o When the depth matches  $d$ , add the node's value to the result.
3. **Steps:**
  - o Initialize the queue with the root node and its depth (0).
  - o Traverse each level of the tree:
    - If a node's depth matches  $d$ , add its value to the result.
    - Enqueue its children with their updated depth.
  - o Return the result list after traversal.
4. **Why it Works:** By explicitly tracking depth, this approach ensures only nodes at the specified depth are included in the result.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Level-Wise Traversal of Trees (Check If Full Binary Tree at Each Level)

**256. Problem: Check if each level of the binary tree is a "full level" (all nodes have 0 or 2 children).**

### Example:

Input: root = [3, 9, 20, null, null, 15, 7]  
 Output: false  
 Explanation: The second level is not full because node 9 has no children.

**Solution:**

```
function isFullBinaryTreePerLevel(root) {  
    if (root === null) {  
        return true;  
    }  
  
    const queue = [root];  
  
    while (queue.length > 0) {  
        const levelSize = queue.length;  
  
        for (let i = 0; i < levelSize; i++) {  
            const node = queue.shift();  
  
            const hasLeft = node.left !== null;  
            const hasRight = node.right !== null;  
  
            // Check if a node has only one child  
            if (hasLeft !== hasRight) {  
                return false;  
            }  
  
            if (hasLeft) {  
                queue.push(node.left);  
            }  
            if (hasRight) {  
                queue.push(node.right);  
            }  
        }  
  
        return true;  
    }  
  
// Test  
const tree = {  
    val: 3,  
    left: { val: 9, left: null, right: null },  
    right: {  
        val: 20,  
        left: { val: 15, left: null, right: null },  
        right: { val: 7, left: null, right: null },  
    },  
};  
  
console.log(isFullBinaryTreePerLevel(tree)); // Output: false
```

### **Explanation:**

1. **Objective:** Determine if each level of the tree is a "full level" where nodes have either 0 or 2 children.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o For each node, check if it has only one child. If so, return false.
  - o Continue processing children of nodes.
3. **Steps:**
  - o Initialize the queue with the root node.
  - o For each node:
    - Check if the left and right children are either both present or both absent.
    - Enqueue its children if it passes the check.
  - o Return true after processing all nodes.
4. **Why it Works:** A full binary tree requires that all nodes have 0 or 2 children, and this condition is validated for each node.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue.

### **Level-Wise Traversal of Trees (List of Leaf Nodes)**

**257. Problem: Return a flat list of all leaf nodes in a binary tree.**

#### **Example:**

```
Input: root = [3, 9, 20, null, null, 15, 7]
Output: [9, 15, 7]
Explanation: Leaf nodes are nodes without children.
```

#### **Solution:**

```
function leafNodes(root) {
  if (root === null) {
    return [];
  }

  const leaves = [];
  const queue = [root];

  while (queue.length > 0) {
    const node = queue.shift();

    if (!node.left && !node.right) {
      leaves.push(node.val);
    }

    if (node.left) {
```

```

        queue.push(node.left);
    }
    if (node.right) {
        queue.push(node.right);
    }
}

return leaves;
}

// Test
const tree = {
    val: 3,
    left: { val: 9, left: null, right: null },
    right: {
        val: 20,
        left: { val: 15, left: null, right: null },
        right: { val: 7, left: null, right: null },
    },
};

console.log(leafNodes(tree)); // Output: [9, 15, 7]

```

### Explanation:

1. **Objective:** Collect all leaf nodes in a binary tree in a flat list.
2. **Approach:**
  - o Use a queue to perform level order traversal.
  - o Identify leaf nodes by checking if both children are null.
  - o Append leaf node values to the leaves list.
3. **Steps:**
  - o Initialize the leaves list and the queue with the root node.
  - o Traverse each node:
    - If it has no children, add its value to the leaves list.
    - Otherwise, enqueue its children.
  - o Return the leaves list after traversal.
4. **Why it Works:** By checking the absence of children for each node, this approach directly identifies leaf nodes.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Level-Wise Traversal of Trees (Deepest Leaves Sum)

**258. Problem:** **Return the sum of the values of the deepest leaves in a binary tree.**

#### Example:

Input: root = [1, 2, 3, 4, 5, null, 6, null, null, 7, 8]

Output: 15

Explanation: The deepest leaves are [7, 8], and their sum is 15.

### Solution:

```
function deepestLeavesSum(root) {  
    if (root === null) {  
        return 0;  
    }  
  
    const queue = [root];  
    let sum = 0;  
  
    while (queue.length > 0) {  
        const levelSize = queue.length;  
        sum = 0; // Reset sum for each level  
  
        for (let i = 0; i < levelSize; i++) {  
            const node = queue.shift();  
            sum += node.val;  
  
            if (node.left) {  
                queue.push(node.left);  
            }  
            if (node.right) {  
                queue.push(node.right);  
            }  
        }  
  
        return sum;  
    }  
  
// Test  
const tree = {  
    val: 1,  
    left: {  
        val: 2,  
        left: { val: 4, left: null, right: null },  
        right: {  
            val: 5,  
            left: { val: 7, left: null, right: null },  
            right: { val: 8, left: null, right: null },  
        },  
    },  
    right: {  
        val: 3,  
    },
```

```

    left: null,
    right: { val: 6, left: null, right: null },
  },
};

console.log(deepestLeavesSum(tree)); // Output: 15

```

### **Explanation:**

1. **Objective:** Calculate the sum of values of the deepest leaves in the binary tree.
2. **Approach:**
  - o Use level order traversal to find the deepest level.
  - o For each level, compute the sum of node values.
  - o Reset the sum for each new level, ensuring the final sum corresponds to the deepest level.
3. **Steps:**
  - o Initialize the queue with the root node.
  - o For each level:
    - Compute the sum of node values and reset it for the next level.
    - Add children of nodes to the queue.
  - o Return the sum after processing all levels.
4. **Why it Works:** By resetting the sum at each level, the final sum corresponds to the deepest level.
5. **Time Complexity:**  $O(n)$  — Each node is visited once.
6. **Space Complexity:**  $O(n)$  — Space for the queue.

### **Level-Wise Traversal of Trees (Nodes at Deepest Level)**

#### **259. Problem: Return all the nodes at the deepest level of a binary tree.**

### **Example:**

```

Input: root = [1, 2, 3, 4, 5, null, 6, null, null, 7, 8]
Output: [7, 8]
Explanation: The deepest level contains the nodes [7, 8].

```

### **Solution:**

```

function deepestLeaves(root) {
  if (root === null) {
    return [];
  }

  const queue = [root];
  let result = [];

  while (queue.length > 0) {

```

```

const levelSize = queue.length;
result = []; // Reset result for each level

for (let i = 0; i < levelSize; i++) {
  const node = queue.shift();
  result.push(node.val);

  if (node.left) {
    queue.push(node.left);
  }
  if (node.right) {
    queue.push(node.right);
  }
}

return result;
}

// Test
const tree = {
  val: 1,
  left: {
    val: 2,
    left: { val: 4, left: null, right: null },
    right: {
      val: 5,
      left: { val: 7, left: null, right: null },
      right: { val: 8, left: null, right: null },
    },
  },
  right: {
    val: 3,
    left: null,
    right: { val: 6, left: null, right: null },
  },
};

console.log(deepestLeaves(tree)); // Output: [7, 8]

```

### Explanation:

1. **Objective:** Collect all nodes at the deepest level of the binary tree.
2. **Approach:**
  - o Use level order traversal to reach the deepest level.
  - o For each level, overwrite the result array with the current level's nodes.
  - o The final result array contains nodes at the deepest level.
3. **Steps:**

- Initialize the queue with the root node.
  - Traverse the tree level by level.
  - At each level:
    - Overwrite the result array with the current level's node values.
    - Enqueue children for the next level.
  - Return the result after processing all levels.
4. **Why it Works:** By overwriting the result array at each level, only the nodes from the deepest level remain at the end.
  5. **Time Complexity:**  $O(n)$  — Each node is visited once.
  6. **Space Complexity:**  $O(n)$  — Space for the queue.

### **Level-Wise Traversal of Trees (Nodes Between Two Levels)**

**260. Problem:** Given the root of a binary tree and two integers low and high, return all nodes between levels low and high.

**Example:**

Input: root = [1, 2, 3, 4, 5, null, 6, null, null, 7, 8], low = 1, high = 2

Output: [[2, 3], [4, 5, 6]]

Explanation:

- Level 1: Nodes are [2, 3].
- Level 2: Nodes are [4, 5, 6].

**Solution:**

```
function nodesBetweenLevels(root, low, high) {
  if (root === null) {
    return [];
  }

  const queue = [root];
  const result = [];
  let level = 0;

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      if (level >= low && level <= high) {
        currentLevel.push(node.val);
      }

      if (node.left) {
        queue.push(node.left);
      }
    }

    if (currentLevel.length > 0) {
      result.push(currentLevel);
    }

    level++;
  }

  return result;
}
```

```

        if (node.right) {
            queue.push(node.right);
        }
    }

    if (currentLevel.length > 0 && level >= low && level <= high) {
        result.push(currentLevel);
    }

    level++;
}

return result;
}

// Test
const tree = {
    val: 1,
    left: {
        val: 2,
        left: { val: 4, left: null, right: null },
        right: {
            val: 5,
            left: { val: 7, left: null, right: null },
            right: { val: 8, left: null, right: null },
        },
    },
    right: {
        val: 3,
        left: null,
        right: { val: 6, left: null, right: null },
    },
};

console.log(nodesBetweenLevels(tree, 1, 2)); // Output: [[2, 3], [4, 5, 6]]

```

### Explanation:

1. **Objective:** Collect nodes between two specified levels in the binary tree.

2. **Approach:**

- Use a level order traversal with a level counter.
- For each level, include nodes only if the level falls within the range [low, high].

3. **Steps:**

- Initialize the queue with the root node and level as 0.
- For each level:
  - Collect nodes if  $\text{low} \leq \text{level} \leq \text{high}$ .
  - Enqueue children for the next level.

- Return the result after processing all levels.
4. **Why it Works:** The level counter ensures nodes from the specified range of levels are included in the result.
  5. **Time Complexity:**  $O(n)$  — Each node is visited once.
  6. **Space Complexity:**  $O(n)$  — Space for the queue and result list.

### Serializing & Restoring Tree Data (Using BFS)

**261. Problem: Design an algorithm to serialize and deserialize a binary tree.**

**Serialization is the process of converting a tree to a single string. Deserialization is the process of converting the string back to the tree.**

**Example:**

```
Input: root = [1, 2, 3, null, null, 4, 5]
Serialization: "1,2,3,null,null,4,5"
Deserialization: Reconstruct the tree from the string.
```

**Solution:**

```
class TreeNode {
    constructor(val, left = null, right = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

```
class Codec {
    // Serialize: Tree to String
    serialize(root) {
        if (root === null) {
            return "";
        }

        const queue = [root];
        const result = [];

        while (queue.length > 0) {
            const node = queue.shift();
            if (node) {
                result.push(node.val);
                queue.push(node.left);
                queue.push(node.right);
            } else {
                result.push("null");
            }
        }
    }

    // Deserialize: String to Tree
    deserialize(data) {
        if (data === "") {
            return null;
        }

        const root = new TreeNode(data[0]);
        const queue = [root];
        let index = 1;

        while (queue.length > 0) {
            const node = queue.shift();
            if (data[index] === "null") {
                node.left = null;
            } else {
                node.left = new TreeNode(data[index]);
                queue.push(node.left);
            }

            if (data[index + 1] === "null") {
                node.right = null;
            } else {
                node.right = new TreeNode(data[index + 1]);
                queue.push(node.right);
            }

            index += 2;
        }

        return root;
    }
}
```

```

        return result.join(",");
    }

// Deserialize: String to Tree
deserialize(data) {
    if (!data) {
        return null;
    }

    const nodes = data.split(",");
    const root = new TreeNode(parseInt(nodes[0]));
    const queue = [root];
    let i = 1;

    while (queue.length > 0) {
        const current = queue.shift();

        if (nodes[i] !== "null") {
            current.left = new TreeNode(parseInt(nodes[i]));
            queue.push(current.left);
        }
        i++;
        if (nodes[i] !== "null") {
            current.right = new TreeNode(parseInt(nodes[i]));
            queue.push(current.right);
        }
        i++;
    }

    return root;
}
}

// Test
const tree = new TreeNode(
    1,
    new TreeNode(2),
    new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized); // Output: "1,2,3,null,null,4,5"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "1,2,3,null,null,4,5"

```

### **Explanation:**

#### **1. Serialization:**

- Use a queue for BFS traversal of the tree.
- Add the value of each node to the result array.
- Add "null" for empty nodes.
- Convert the result array to a comma-separated string.

#### **2. Deserialization:**

- Split the serialized string into an array of node values.
- Create the root node using the first value.
- Use a queue to reconstruct the tree level by level.
- Assign left and right children for each node based on the array.

#### **3. Edge Cases:**

- Empty tree: Return an empty string for serialization and null for deserialization.
- Single-node tree: Correctly handle trees with only one node.

#### **4. Time Complexity:**

- Serialization:  $O(n)$ , where  $n$  is the number of nodes in the tree.
- Deserialization:  $O(n)$ .

#### **5. Space Complexity:**

- Serialization:  $O(n)$  for the result array.
- Deserialization:  $O(n)$  for the queue.

### **Serializing & Restoring Tree Data (Using DFS - Preorder Traversal)**

**262. Problem: Design an algorithm to serialize and deserialize a binary tree using Depth-First Search (DFS) with preorder traversal.**

#### **Example:**

```
Input: root = [1, 2, 3, null, null, 4, 5]
Serialization: "1,2,null,null,3,4,null,null,5,null,null"
Deserialization: Reconstruct the tree from the string.
```

#### **Solution:**

```
class TreeNode {
    constructor(val, left = null, right = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Codec {
    // Serialize: Tree to String using Preorder Traversal
    serialize(root) {
        function dfs(node) {
            if (node === null) return '';
            return `${node.val}, ${dfs(node.left)}, ${dfs(node.right)}  
`;
        }
        return dfs(root);
    }

    // Deserialize: String to Tree using Preorder Traversal
    deserialize(data) {
        const nodes = data.split(', ');
        const root = new TreeNode(nodes[0]);
        const stack = [root];
        let i = 1;

        for (let j = 1; j < nodes.length; j++) {
            const node = stack[i - 1];
            if (nodes[j] === 'null') {
                node.left = null;
            } else {
                node.left = new TreeNode(nodes[j]);
                stack.push(node.left);
            }
            i++;
            if (i === stack.length) {
                stack.pop();
                i--;
            }
        }
        return root;
    }
}
```

```

if (node === null) {
    result.push("null");
    return;
}
result.push(node.val);
dfs(node.left);
dfs(node.right);
}

const result = [];
dfs(root);
return result.join(",");
}

// Deserialize: String to Tree using Preorder Traversal
deserialize(data) {
    const nodes = data.split(",");
    let index = 0;

    function buildTree() {
        if (nodes[index] === "null") {
            index++;
            return null;
        }

        const node = new TreeNode(parseInt(nodes[index]));
        index++;
        node.left = buildTree();
        node.right = buildTree();
        return node;
    }

    return buildTree();
}
}

// Test
const tree = new TreeNode(
    1,
    new TreeNode(2),
    new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized); // Output: "1,2,null,null,3,4,null,null,5,null,null"

const deserialized = codec.deserialize(serialized);

```

```
console.log(codec.serialize(deserialized)); // Output: "1,2,null,null,3,4,null,null,5,null,null"
```

### Explanation:

1. **Serialization (Preorder DFS):**
  - o Traverse the tree in preorder: root → left → right.
  - o Append the value of each node to the result array.
  - o Add "null" for empty nodes to preserve structure.
2. **Deserialization:**
  - o Split the serialized string into an array of node values.
  - o Use a recursive helper function to rebuild the tree.
  - o At each step:
    - If the current value is "null", return null.
    - Otherwise, create a new node with the current value and recursively build its left and right subtrees.
3. **Edge Cases:**
  - o Empty tree: Return an empty string for serialization and null for deserialization.
  - o Single-node tree: Correctly handle trees with only one node.
4. **Time Complexity:**
  - o Serialization: O(n), where n is the number of nodes in the tree.
  - o Deserialization: O(n).
5. **Space Complexity:**
  - o Serialization: O(n) for the result array.
  - o Deserialization: O(n) for the recursion stack.

### Serializing & Restoring Tree Data (Using Postorder Traversal)

**263. Problem: Design an algorithm to serialize and deserialize a binary tree using Depth-First Search (DFS) with postorder traversal.**

#### Solution:

```
class Codec {  
    // Serialize: Tree to String using Postorder Traversal  
    serialize(root) {  
        function dfs(node) {  
            if (node === null) {  
                result.push("null");  
                return;  
            }  
            dfs(node.left);  
            dfs(node.right);  
            result.push(node.val);  
        }  
  
        const result = [];  
        dfs(root);  
        return result.join(",");  
    }  
}
```

```

}

// Deserialize: String to Tree using Postorder Traversal
deserialize(data) {
  const nodes = data.split(",");
  let index = nodes.length - 1;

  function buildTree() {
    if (nodes[index] === "null") {
      index--;
      return null;
    }

    const node = new TreeNode(parseInt(nodes[index]));
    index--;
    node.right = buildTree();
    node.left = buildTree();
    return node;
  }

  return buildTree();
}

// Test
const tree = new TreeNode(
  1,
  new TreeNode(2),
  new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized); // Output: "null,null,2,null,null,4,null,null,5,3,1"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "null,null,2,null,null,4,null,null,5,3,1"

```

### Explanation:

#### 1. Serialization (Postorder DFS):

- Traverse the tree in postorder: left → right → root.
- Append the value of each node to the result array.
- Add "null" for empty nodes to preserve structure.

#### 2. Deserialization:

- Split the serialized string into an array of node values.
- Use a recursive helper function to rebuild the tree in reverse postorder.
- At each step:

- If the current value is "null", return null.
- Otherwise, create a new node with the current value and recursively build its right and left subtrees.

3. **Time Complexity:** O(n)
4. **Space Complexity:** O(n)

### **Serializing & Restoring Tree Data (Space-Optimized Using Minimal Nulls)**

**264. Problem: Optimize the serialization of a binary tree by reducing the number of "null" placeholders, aiming for minimal space usage.**

**Solution:**

```
class TreeNode {
    constructor(val, left = null, right = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Codec {
    // Serialize: Space-optimized Tree to String
    serialize(root) {
        function dfs(node) {
            if (node === null) {
                return;
            }
            result.push(node.val);
            if (node.left || node.right) {
                result.push(node.left ? node.left.val : "null");
                result.push(node.right ? node.right.val : "null");
            }
            dfs(node.left);
            dfs(node.right);
        }

        const result = [];
        dfs(root);
        return result.join(",");
    }

    // Deserialize: String to Tree
    deserialize(data) {
        if (!data) return null;

        const values = data.split(",");
        const root = new TreeNode(parseInt(values[0]));
        const queue = [root];

```

```

let i = 1;

while (queue.length > 0) {
  const node = queue.shift();
  if (values[i] !== "null") {
    node.left = new TreeNode(parseInt(values[i]));
    queue.push(node.left);
  }
  i++;
  if (values[i] !== "null") {
    node.right = new TreeNode(parseInt(values[i]));
    queue.push(node.right);
  }
  i++;
}

return root;
}

// Test
const tree = new TreeNode(
  1,
  new TreeNode(2),
  new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized); // Output: "1,2,3,null,null,4,5"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "1,2,3,null,null,4,5"

```

### Explanation:

1. **Serialization (Space-Optimized):**
  - o Use DFS preorder traversal.
  - o Only append "null" if a node has at least one child, reducing the number of "null" values.
  - o The output string is shorter and preserves the tree structure.
2. **Deserialization:**
  - o Use a queue to reconstruct the tree.
  - o Iterate through the serialized data:
    - If a value is not "null", create a new node and attach it to the current parent.
    - Enqueue the new nodes for further processing.
3. **Edge Cases:**

- Empty tree: Return an empty string for serialization and null for deserialization.
- Single-node tree: Handle trees with one node correctly.

#### 4. Time Complexity:

- Serialization:  $O(n)$ , where  $n$  is the number of nodes in the tree.
- Deserialization:  $O(n)$ .

#### 5. Space Complexity:

- Serialization:  $O(n)$  for the result array.
- Deserialization:  $O(n)$  for the queue.

### Serialize and Deserialize Complete Binary Tree

**265. Problem: Design a specialized algorithm to serialize and deserialize a complete binary tree for better efficiency.**

#### Solution:

```
class Codec {
    // Serialize: Complete Binary Tree to String
    serialize(root) {
        if (root === null) return "";

        const result = [];
        const queue = [root];

        while (queue.length > 0) {
            const node = queue.shift();
            if (node) {
                result.push(node.val);
                queue.push(node.left);
                queue.push(node.right);
            } else {
                result.push("null");
            }
        }

        // Remove trailing "null" values
        while (result[result.length - 1] === "null") {
            result.pop();
        }

        return result.join(",");
    }

    // Deserialize: String to Complete Binary Tree
    deserialize(data) {
        if (!data) return null;

        const values = data.split(",");
        let index = 0;
        let root = new TreeNode(values[0]);
        let currentLevel = [root];
        let nextLevel = [];

        for (let i = 1; i < values.length; i++) {
            if (values[i] !== "null") {
                const node = new TreeNode(values[i]);
                currentLevel.push(node);
                nextLevel.push(node);
            }
        }
    }
}
```

```

const root = new TreeNode(parseInt(values[0]));
const queue = [root];
let i = 1;

while (i < values.length) {
  const node = queue.shift();

  if (values[i] !== "null") {
    node.left = new TreeNode(parseInt(values[i]));
    queue.push(node.left);
  }
  i++;
}

if (i < values.length && values[i] !== "null") {
  node.right = new TreeNode(parseInt(values[i]));
  queue.push(node.right);
}
i++;
}

return root;
}

// Test
const tree = new TreeNode(
  1,
  new TreeNode(2),
  new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized); // Output: "1,2,3,4,5"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "1,2,3,4,5"

```

### Explanation:

1. **Serialization for Complete Binary Tree:**
  - o Use BFS traversal.
  - o Remove trailing "null" values from the result to save space.
2. **Deserialization for Complete Binary Tree:**
  - o Use a queue to reconstruct the tree level by level.
  - o Reconstruct nodes left-to-right from the serialized data.
3. **Why It's Efficient:**

- A complete binary tree minimizes the number of "null" placeholders, reducing space complexity.
4. **Time Complexity:**
- Serialization:  $O(n)$ .
  - Deserialization:  $O(n)$ .
5. **Space Complexity:**
- Serialization:  $O(n)$ .
  - Deserialization:  $O(n)$ .

### **Serialize and Deserialize Binary Search Tree (BST)**

**266. Problem: Design an algorithm to serialize and deserialize a binary search tree (BST). Leverage the BST property for efficient serialization and deserialization.**

**Solution:**

```
class TreeNode {
    constructor(val, left = null, right = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Codec {
    // Serialize: BST to String using Preorder Traversal
    serialize(root) {
        function preorder(node) {
            if (node === null) {
                return [];
            }
            return [node.val, ...preorder(node.left), ...preorder(node.right)];
        }

        return preorder(root).join(",");
    }

    // Deserialize: String to BST
    deserialize(data) {
        if (!data) return null;

        const preorder = data.split(",").map(Number);

        function buildTree(lower, upper) {
            if (preorder.length === 0 || preorder[0] < lower || preorder[0] > upper) {
                return null;
            }

            const value = preorder.shift();

```

```

const node = new TreeNode(value);
node.left = buildTree(lower, value);
node.right = buildTree(value, upper);
return node;
}

return buildTree(-Infinity, Infinity);
}
}

// Test
const bst = new TreeNode(
  8,
  new TreeNode(3, new TreeNode(1), new TreeNode(6)),
  new TreeNode(10, null, new TreeNode(14))
);
const codec = new Codec();

const serialized = codec.serialize(bst);
console.log(serialized); // Output: "8,3,1,6,10,14"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "8,3,1,6,10,14"

```

### Explanation:

#### 1. Serialization:

- Traverse the BST in preorder (root → left → right).
- Store the node values in an array and convert it to a comma-separated string.
- Preorder traversal ensures the tree structure is preserved.

#### 2. Deserialization:

- Convert the serialized string back into an array of numbers (preorder array).
- Use the BST property to rebuild the tree recursively:
  - For each node, ensure that its value is within the range [lower, upper].
  - Construct the left subtree using values smaller than the current node.
  - Construct the right subtree using values larger than the current node.

#### 3. Advantages:

- The BST property allows for efficient deserialization without requiring "null" placeholders.

#### 4. Edge Cases:

- Empty tree: Return an empty string for serialization and null for deserialization.
- Single-node tree: Handle trees with one node correctly.

#### 5. Time Complexity:

- Serialization:  $O(n)$ , where  $n$  is the number of nodes in the tree.
- Deserialization:  $O(n)$ , as each node is processed once.

#### 6. Space Complexity:

- Serialization:  $O(n)$  for the result array.

- o Deserialization: O(n) for the recursion stack.

### **Serialize and Deserialize N-ary Tree**

**267. Problem: Design an algorithm to serialize and deserialize an N-ary tree, where each node can have an arbitrary number of children.**

**Solution:**

```
class NaryTreeNode {
  constructor(val, children = []) {
    this.val = val;
    this.children = children;
  }
}

class Codec {
  // Serialize: N-ary Tree to String
  serialize(root) {
    function dfs(node) {
      if (node === null) return "";
      const childrenCount = node.children.length;
      const serializedChildren = node.children.map(dfs).join(",");
      return `${node.val},${childrenCount},${serializedChildren}`;
    }
    return dfs(root);
  }

  // Deserialize: String to N-ary Tree
  deserialize(data) {
    if (!data) return null;

    const values = data.split(",");
    let index = 0;

    function buildTree() {
      if (index >= values.length) return null;

      const value = parseInt(values[index++]);
      const childrenCount = parseInt(values[index++]);
      const node = new NaryTreeNode(value);

      for (let i = 0; i < childrenCount; i++) {
        node.children.push(buildTree());
      }
      return node;
    }

    return buildTree();
  }
}
```

```

    }

}

// Test
const naryTree = new NaryTreeNode(1, [
  new NaryTreeNode(2),
  new NaryTreeNode(3, [new NaryTreeNode(6), new NaryTreeNode(7)]),
  new NaryTreeNode(4),
  new NaryTreeNode(5),
]);
const codec = new Codec();

const serialized = codec.serialize(naryTree);
console.log(serialized); // Output: "1,4,2,0,3,2,6,0,7,0,4,0,5,0"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "1,4,2,0,3,2,6,0,7,0,4,0,5,0"

```

### **Explanation:**

#### **1. Serialization:**

- Traverse the N-ary tree in preorder (root → children).
- For each node, append its value and the number of its children.
- Serialize each child recursively.

#### **2. Deserialization:**

- Split the serialized string into an array of values.
- Use a recursive helper function to rebuild the tree:
  - Create a node with the current value.
  - Use the number of children to reconstruct the subtree.

#### **3. Edge Cases:**

- Empty tree: Return an empty string for serialization and null for deserialization.
- Single-node tree: Handle trees with one node correctly.

#### **4. Time Complexity:**

- Serialization: O(n), where n is the number of nodes.
- Deserialization: O(n).

#### **5. Space Complexity:**

- Serialization: O(n) for the result string.
- Deserialization: O(n) for the recursion stack.

### **Serializing & Restoring Tree Data (Iterative Approach Using Stack)**

#### **268. Problem: Design an algorithm to serialize and deserialize a binary tree using an iterative approach.**

### **Solution:**

```

class TreeNode {
  constructor(val, left = null, right = null) {

```

```

    this.val = val;
    this.left = left;
    this.right = right;
}
}

class Codec {
// Serialize: Tree to String using Iterative Preorder
serialize(root) {
    if (root === null) return "";

    const stack = [root];
    const result = [];

    while (stack.length > 0) {
        const node = stack.pop();
        if (node) {
            result.push(node.val);
            stack.push(node.right); // Push right first to process left first
            stack.push(node.left);
        } else {
            result.push("null");
        }
    }

    return result.join(",");
}

// Deserialize: String to Tree using Iterative Preorder
deserialize(data) {
    if (!data) return null;

    const values = data.split(",");
    const root = new TreeNode(parseInt(values[0]));
    const stack = [root];
    let index = 1;

    while (stack.length > 0) {
        const node = stack.pop();

        if (values[index] !== "null") {
            node.left = new TreeNode(parseInt(values[index]));
            stack.push(node.left);
        }
        index++;

        if (values[index] !== "null") {
            node.right = new TreeNode(parseInt(values[index]));
        }
    }

    return root;
}
}

```

```

        stack.push(node.right);
    }
    index++;
}

return root;
}
}

// Test
const tree = new TreeNode(
  1,
  new TreeNode(2),
  new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized); // Output: "1,2,null,null,3,4,null,null,5,null,null"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "1,2,null,null,3,4,null,null,5,null,null"

```

### **Explanation:**

#### **1. Serialization (Iterative Preorder):**

- Use a stack to simulate the recursive preorder traversal (root → left → right).
- Push the right child first so that the left child is processed first.
- Append "null" for empty nodes to preserve tree structure.

#### **2. Deserialization (Iterative):**

- Use the serialized string to create nodes iteratively.
- Maintain a stack to keep track of nodes whose children need to be constructed.
- Add left and right children in order, popping nodes from the stack once their children are assigned.

#### **3. Edge Cases:**

- Empty tree: Return an empty string for serialization and null for deserialization.
- Single-node tree: Handle trees with one node correctly.

#### **4. Time Complexity:**

- Serialization: O(n), where n is the number of nodes.
- Deserialization: O(n).

#### **5. Space Complexity:**

- Serialization: O(n) for the result array.
- Deserialization: O(n) for the stack.

### **Serializing & Restoring Tree Data (Inorder and Preorder)**

**269. Problem: Design an algorithm to serialize and deserialize a binary tree using both inorder and preorder traversals.**

**Solution:**

```
class Codec {  
    // Serialize: Tree to Inorder and Preorder Strings  
    serialize(root) {  
        function inorder(node) {  
            if (!node) return [];  
            return [...inorder(node.left), node.val, ...inorder(node.right)];  
        }  
  
        function preorder(node) {  
            if (!node) return [];  
            return [node.val, ...preorder(node.left), ...preorder(node.right)];  
        }  
  
        const inorderTraversal = inorder(root);  
        const preorderTraversal = preorder(root);  
  
        return {  
            inorder: inorderTraversal.join(","),  
            preorder: preorderTraversal.join(","),  
        };  
    }  
  
    // Deserialize: Inorder and Preorder Strings to Tree  
    deserialize(data) {  
        if (!data.inorder || !data.preorder) return null;  
  
        const inorder = data.inorder.split(",").map(Number);  
        const preorder = data.preorder.split(",").map(Number);  
  
        function buildTree(preStart, preEnd, inStart, inEnd) {  
            if (preStart > preEnd || inStart > inEnd) return null;  
  
            const rootVal = preorder[preStart];  
            const root = new TreeNode(rootVal);  
  
            const inIndex = inorder.indexOf(rootVal);  
            const leftSize = inIndex - inStart;  
  
            root.left = buildTree(  
                preStart + 1,  
                preStart + leftSize,  
                inStart,  
                inIndex - 1  
            );  
        }  
    }  
}
```

```

root.right = buildTree(
    preStart + leftSize + 1,
    preEnd,
    inIndex + 1,
    inEnd
);

return root;
}

return buildTree(0, preorder.length - 1, 0, inorder.length - 1);
}
}

// Test
const tree = new TreeNode(
    1,
    new TreeNode(2),
    new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized);
// Output: { inorder: "2,1,4,3,5", preorder: "1,2,3,4,5" }

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized));
// Output: { inorder: "2,1,4,3,5", preorder: "1,2,3,4,5" }

```

## Explanation:

### 1. Serialization:

- Traverse the tree in inorder and preorder and store both traversals.
- These traversals uniquely identify a binary tree.

### 2. Deserialization:

- Use the preorder traversal to locate the root.
- Use the inorder traversal to determine the left and right subtrees.
- Recursively rebuild the tree.

### 3. Edge Cases:

- Empty tree: Return empty strings for traversals.
- Single-node tree: Handle correctly with one node.

### 4. Time Complexity:

- Serialization:  $O(n)$ .
- Deserialization:  $O(n)$ .

### 5. Space Complexity:

- Serialization:  $O(n)$  for the result strings.
- Deserialization:  $O(n)$  for the recursion stack.

## Serializing & Restoring Tree Data (Level Order + Markers for Null Nodes)

**270. Problem: Design an algorithm to serialize and deserialize a binary tree using level order traversal, including explicit markers for null nodes.**

**Solution:**

```
class TreeNode {  
    constructor(val, left = null, right = null) {  
        this.val = val;  
        this.left = left;  
        this.right = right;  
    }  
}  
  
class Codec {  
    // Serialize: Tree to String using Level Order  
    serialize(root) {  
        if (root === null) return "";  
  
        const result = [];  
        const queue = [root];  
  
        while (queue.length > 0) {  
            const node = queue.shift();  
            if (node) {  
                result.push(node.val);  
                queue.push(node.left);  
                queue.push(node.right);  
            } else {  
                result.push("null");  
            }  
        }  
  
        return result.join(",");  
    }  
  
    // Deserialize: String to Tree using Level Order  
    deserialize(data) {  
        if (!data) return null;  
  
        const values = data.split(",");  
        const root = new TreeNode(parseInt(values[0]));  
        const queue = [root];  
        let index = 1;  
  
        while (queue.length > 0) {  
            const node = queue.shift();
```

```

    if (values[index] !== "null") {
      node.left = new TreeNode(parseInt(values[index]));
      queue.push(node.left);
    }
    index++;
  }

  if (values[index] !== "null") {
    node.right = new TreeNode(parseInt(values[index]));
    queue.push(node.right);
  }
  index++;
}

return root;
}
}

// Test
const tree = new TreeNode(
  1,
  new TreeNode(2),
  new TreeNode(3, new TreeNode(4), new TreeNode(5))
);
const codec = new Codec();

const serialized = codec.serialize(tree);
console.log(serialized); // Output: "1,2,3,null,null,4,5"

const deserialized = codec.deserialize(serialized);
console.log(codec.serialize(deserialized)); // Output: "1,2,3,null,null,4,5"

```

### Explanation:

1. **Serialization:**
  - o Perform level order traversal using a queue.
  - o Append "null" for missing nodes to ensure the structure is preserved.
  - o Convert the resulting list into a comma-separated string.
2. **Deserialization:**
  - o Split the serialized string into an array of node values.
  - o Use a queue to reconstruct the tree level by level.
  - o Create nodes for non-null values and attach them as children of the current node.
3. **Edge Cases:**
  - o Empty tree: Return an empty string for serialization and null for deserialization.
  - o Single-node tree: Handle correctly.
4. **Time Complexity:**
  - o Serialization: O(n), where n is the number of nodes in the tree.

- o Deserialization: O(n).

## 5. Space Complexity:

- o Serialization: O(n) for the result array.
- o Deserialization: O(n) for the queue.

### Checking Subtree Presence

**271. Problem:** You are given two binary trees root and subRoot. Determine if subRoot is a subtree of root.

A subtree of a binary tree T is a tree that consists of a node in T and all of its descendants. The subtree T can also be considered as a subtree of itself.

**Example:**

```
Input: root = [3, 4, 5, 1, 2], subRoot = [4, 1, 2]
Output: true
Explanation: The subtree rooted at 4 in the main tree matches `subRoot`.
```

**Solution:**

```
class TreeNode {
    constructor(val, left = null, right = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

function isSubtree(root, subRoot) {
    if (!root) return false;
    if (isSameTree(root, subRoot)) return true;
    return isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);
}

function isSameTree(tree1, tree2) {
    if (!tree1 && !tree2) return true; // Both are null
    if (!tree1 || !tree2) return false; // One is null
    if (tree1.val !== tree2.val) return false; // Values don't match

    // Recursively check left and right subtrees
    return (
        isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right)
    );
}

// Test
const root = new TreeNode(
    3,
```

```

new TreeNode(4, new TreeNode(1), new TreeNode(2)),
new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtree(root, subRoot)); // Output: true

```

### **Explanation:**

1. **Main Function (isSubtree):**
  - o Traverse the root tree.
  - o At each node, check if it is identical to the subRoot using the helper function isSameTree.
  - o If a match is found, return true. Otherwise, recursively check the left and right subtrees.
2. **Helper Function (isSameTree):**
  - o Compare two trees node by node:
    - If both nodes are null, they are identical.
    - If one node is null and the other is not, they are not identical.
    - If the values of the nodes are different, they are not identical.
  - o Recursively check the left and right children.
3. **Base Cases:**
  - o If root is null, return false because a null tree cannot contain any subtree.
  - o If both trees are empty (null), they are identical.
4. **Time Complexity:**
  - o For each node in root, the isSameTree function is called.
  - o Worst case:  $O(m * n)$ , where m is the number of nodes in root and n is the number of nodes in subRoot.
5. **Space Complexity:**
  - o  $O(h)$ , where h is the height of the tree (due to the recursion stack).

### **Checking Subtree Presence (Optimized Using String Representations)**

**272. Problem:** Given two binary trees root and subRoot, determine if subRoot is a subtree of root. Optimize the solution by leveraging string representations of trees.

### **Solution:**

```

function isSubtreeUsingStrings(root, subRoot) {
  function serialize(node) {
    if (!node) return "null";
    return `#${node.val} ${serialize(node.left)} ${serialize(node.right)} `;
  }

  const rootString = serialize(root);
  const subRootString = serialize(subRoot);

  return rootString.includes(subRootString);
}

```

```
// Test
const root = new TreeNode(
  3,
  new TreeNode(4, new TreeNode(1), new TreeNode(2)),
  new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeUsingStrings(root, subRoot)); // Output: true
```

### **Explanation:**

#### **1. Serialization (serialize):**

- Perform a preorder traversal (root → left → right) of the tree.
- Add a marker (e.g., "null") for empty nodes to preserve the structure.
- Concatenate the node values and markers into a single string.

#### **2. Checking Subtree:**

- Convert both root and subRoot into serialized strings.
- Use the .includes() method to check if the serialized subRoot is a substring of root.

#### **3. Advantages of String Representation:**

- Avoids repeatedly comparing subtrees directly.
- Efficient substring matching using native string operations.

#### **4. Edge Cases:**

- root or subRoot is null: Handle gracefully.
- Large trees: Handles efficiently with substring checks.

#### **5. Time Complexity:**

- Serialization:  $O(n)$  for root and  $O(m)$  for subRoot, where n and m are the number of nodes in the respective trees.
- Substring Check:  $O(n + m)$ .
- Total:  $O(n + m)$ .

#### **6. Space Complexity:**

- $O(n + m)$  for the serialized strings.

### **Checking Subtree Presence (Using Hashing for Optimization)**

#### **273. Problem: Use a hashing approach to optimize subtree detection.**

### **Solution:**

```
function isSubtreeUsingHashing(root, subRoot) {
  const subtreeHashes = new Set();

  function hash(node) {
    if (!node) return "null";

    const leftHash = hash(node.left);
    const rightHash = hash(node.right);
```

```

const currentHash = `${node.val},${leftHash},${rightHash}`;

subtreeHashes.add(currentHash);
return currentHash;
}

function checkSubtree(node) {
  if (!node) return false;
  const subRootHash = hash(subRoot);
  return subtreeHashes.has(subRootHash);
}

hash(root);
return checkSubtree(root);
}

// Test
const root = new TreeNode(
  3,
  new TreeNode(4, new TreeNode(1), new TreeNode(2)),
  new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeUsingHashing(root, subRoot)); // Output: true

```

### Explanation:

- 1. Hash Generation (hash):**
  - Recursively compute a unique hash for each subtree by combining the hashes of the left and right children and the current node's value.
  - Store all subtree hashes in a set for quick lookup.
- 2. Checking Subtree:**
  - Compute the hash of subRoot.
  - Check if the subRoot hash exists in the set of hashes for root.
- 3. Advantages:**
  - Avoids redundant subtree comparisons.
  - Efficient lookup using a hash set.
- 4. Edge Cases:**
  - Empty trees: Handle null trees gracefully.
  - Repeated structures: Hashing avoids duplicate comparisons.
- 5. Time Complexity:**
  - Hashing root:  $O(n)$ .
  - Hashing subRoot:  $O(m)$ .
  - Total:  $O(n + m)$ .
- 6. Space Complexity:**
  - $O(n + m)$  for the hash set and recursion stack.

## Checking Subtree Presence (Iterative Approach)

**274. Problem:** Implement an iterative solution to determine if subRoot is a subtree of root.

**Solution:**

```
function isSubtreeIterative(root, subRoot) {  
    if (!root) return false;  
  
    const stack = [root];  
  
    while (stack.length > 0) {  
        const node = stack.pop();  
  
        if (node.val === subRoot.val && isSameTree(node, subRoot)) {  
            return true;  
        }  
  
        if (node.right) stack.push(node.right);  
        if (node.left) stack.push(node.left);  
    }  
  
    return false;  
}  
  
function isSameTree(tree1, tree2) {  
    if (!tree1 && !tree2) return true;  
    if (!tree1 || !tree2) return false;  
    if (tree1.val !== tree2.val) return false;  
  
    return (  
        isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right)  
    );  
}  
  
// Test  
const root = new TreeNode(  
    3,  
    new TreeNode(4, new TreeNode(1), new TreeNode(2)),  
    new TreeNode(5)  
);  
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));  
  
console.log(isSubtreeIterative(root, subRoot)); // Output: true
```

**Explanation:**

### 1. Iterative Traversal:

- o Use a stack to perform a depth-first traversal of root.
- o At each node, compare it with subRoot using the helper function isSameTree.

### 2. Advantages:

- o Avoids recursion, which can be beneficial for large trees.
- o Efficient for small subtrees.

### 3. Edge Cases:

- o Empty root or subRoot.
- o Identical trees: Should return true.

### 4. Time Complexity:

- o  $O(n * m)$  in the worst case.

### 5. Space Complexity:

- o  $O(h)$ , where h is the height of the tree.

## Checking Subtree Presence (Optimized Recursive Solution with Early Exit)

**275. Problem: Given two binary trees root and subRoot, determine if subRoot is a subtree of root. Use an optimized recursive solution with early exit to reduce redundant checks.**

### Solution:

```
function isSubtreeOptimized(root, subRoot) {  
    if (!root) return false; // Base case: main tree is empty  
    if (isSameTree(root, subRoot)) return true; // Check if current trees match  
    return (  
        isSubtreeOptimized(root.left, subRoot) ||  
        isSubtreeOptimized(root.right, subRoot)  
    ); // Recurse  
}  
  
function isSameTree(tree1, tree2) {  
    if (!tree1 && !tree2) return true; // Both are null  
    if (!tree1 || !tree2) return false; // One is null  
    if (tree1.val !== tree2.val) return false; // Values don't match  
  
    return (  
        isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right)  
    ); // Check subtrees  
}  
  
// Test  
const root = new TreeNode(  
    3,  
    new TreeNode(4, new TreeNode(1), new TreeNode(2)),  
    new TreeNode(5)  
);  
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));
```

```
console.log(isSubtreeOptimized(root, subRoot)); // Output: true
```

### Explanation:

1. **Recursive Traversal:**
  - o Traverse the root tree recursively.
  - o For each node, check if it matches the structure and values of subRoot.
2. **Early Exit:**
  - o As soon as a match is found, return true.
  - o If no match is found at a particular node, recursively check the left and right subtrees.
3. **Advantages:**
  - o Reduces redundant comparisons by halting early when possible.
  - o Efficient for trees with many mismatches or large subtrees.
4. **Edge Cases:**
  - o root or subRoot is null.
  - o subRoot is identical to root.
5. **Time Complexity:**
  - o  $O(m * n)$ , where m is the number of nodes in root and n is the number of nodes in subRoot.
6. **Space Complexity:**
  - o  $O(h)$ , where h is the height of the tree (due to recursion stack).

### Checking Subtree Presence (Using Postorder Traversal)

**276. Problem: Determine if subRoot is a subtree of root using a postorder traversal approach.**

#### Solution:

```
function isSubtreePostorder(root, subRoot) {  
    if (!root) return false; // Base case: main tree is empty  
    return (  
        isSameTreePostorder(root, subRoot) ||  
        isSubtreePostorder(root.left, subRoot) ||  
        isSubtreePostorder(root.right, subRoot)  
    );  
}  
  
function isSameTreePostorder(tree1, tree2) {  
    if (!tree1 && !tree2) return true; // Both are null  
    if (!tree1 || !tree2) return false; // One is null  
    if (tree1.val !== tree2.val) return false; // Values don't match  
  
    return (  
        isSameTreePostorder(tree1.left, tree2.left) &&  
        isSameTreePostorder(tree1.right, tree2.right)  
    );  
}
```

```
// Test
const root = new TreeNode(
  3,
  new TreeNode(4, new TreeNode(1), new TreeNode(2)),
  new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreePostorder(root, subRoot)); // Output: true
```

### **Explanation:**

#### **1. Postorder Traversal:**

- o Check the left and right subtrees before comparing the current root node.
- o This ensures that the structure of the entire subtree is validated before moving up.

#### **2. Edge Cases:**

- o Empty root or subRoot.
- o Fully matching or mismatching trees.

#### **3. Time Complexity:**

- o  $O(m * n)$ , where m is the number of nodes in root and n is the number of nodes in subRoot.

#### **4. Space Complexity:**

- o  $O(h)$ , where h is the height of the tree.

### **Checking Subtree Presence (Serialize Both Trees and Compare)**

**277. Problem: Use serialization to convert both root and subRoot into strings and compare them directly.**

### **Solution:**

```
function isSubtreeSerialized(root, subRoot) {
  function serialize(node) {
    if (!node) return "null";
    return `(${serialize(node.left)})${node.val}(${serialize(node.right)})`;
  }

  const rootSerialized = serialize(root);
  const subRootSerialized = serialize(subRoot);

  return rootSerialized.includes(subRootSerialized);
}

// Test
const root = new TreeNode(
  3,
```

```

new TreeNode(4, new TreeNode(1), new TreeNode(2)),
new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeSerialized(root, subRoot)); // Output: true

```

### **Explanation:**

1. **Serialization:**
  - o Perform a recursive traversal of both root and subRoot to convert them into serialized strings.
  - o Use parentheses to preserve structure.
2. **Comparison:**
  - o Check if the serialized subRoot string is a substring of the serialized root string.
3. **Advantages:**
  - o Simplifies subtree detection by leveraging string operations.
4. **Edge Cases:**
  - o Empty trees.
  - o Identical root and subRoot.
5. **Time Complexity:**
  - o Serialization:  $O(n)$  for root and  $O(m)$  for subRoot.
  - o Substring check:  $O(n + m)$ .
  - o Total:  $O(n + m)$ .
6. **Space Complexity:**
  - o  $O(n + m)$  for the serialized strings.

### **Checking Subtree Presence (Iterative BFS Approach)**

**278. Problem: Determine if subRoot is a subtree of root using an iterative BFS approach for traversal.**

### **Solution:**

```

function isSubtreeBFS(root, subRoot) {
  if (!root) return false;

  const queue = [root];

  while (queue.length > 0) {
    const current = queue.shift();

    if (isSameTree(current, subRoot)) {
      return true;
    }

    if (current.left) {
      queue.push(current.left);
    }
  }
}

function isSameTree(tree1, tree2) {
  if (!tree1 || !tree2) return tree1 === tree2;

  if (tree1.val !== tree2.val) return false;

  return isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right);
}

```

```

    }

    if (current.right) {
        queue.push(current.right);
    }
}

return false;
}

function isSameTree(tree1, tree2) {
    if (!tree1 && !tree2) return true; // Both are null
    if (!tree1 || !tree2) return false; // One is null
    if (tree1.val !== tree2.val) return false; // Values don't match

    // Recursively compare left and right subtrees
    return (
        isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right)
    );
}

// Test
const root = new TreeNode(
    3,
    new TreeNode(4, new TreeNode(1), new TreeNode(2)),
    new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeBFS(root, subRoot)); // Output: true

```

### Explanation:

#### 1. BFS Traversal:

- Use a queue to traverse root level by level.
- At each node, check if it matches subRoot using the isSameTree helper.

#### 2. Helper Function (isSameTree):

- Compare two trees recursively:
  - If both nodes are null, they are identical.
  - If one node is null or their values differ, they are not identical.
  - Recursively check left and right subtrees.

#### 3. Advantages:

- Iterative traversal avoids recursion stack overhead.
- Efficient for balanced trees or when subRoot is close to the top of root.

#### 4. Edge Cases:

- root or subRoot is null.
- subRoot is identical to root.

#### 5. Time Complexity:

- o  $O(m * n)$ , where m is the number of nodes in root and n is the number of nodes in subRoot.

## 6. Space Complexity:

- o  $O(h)$ , where h is the height of the tree (queue size).

### Checking Subtree Presence (Using Inorder Traversal and Preorder Traversal)

**279. Problem:** Leverage both **inorder traversal** and **preorder traversal** to check if subRoot is a subtree of root.

**Solution:**

```
function isSubtreeInorderPreorder(root, subRoot) {
    function serializePreorder(node) {
        if (!node) return "null";
        return `${node.val},${serializePreorder(node.left)},${serializePreorder(
            node.right
        )}`;
    }
}

function serializeInorder(node) {
    if (!node) return "null";
    return `${serializeInorder(node.left)},${node.val},${serializeInorder(
        node.right
    )}`;
}

const rootPreorder = serializePreorder(root);
const rootInorder = serializeInorder(root);
const subRootPreorder = serializePreorder(subRoot);
const subRootInorder = serializeInorder(subRoot);

return (
    rootPreorder.includes(subRootPreorder) &&
    rootInorder.includes(subRootInorder)
);
}

// Test
const root = new TreeNode(
    3,
    new TreeNode(4, new TreeNode(1), new TreeNode(2)),
    new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeInorderPreorder(root, subRoot)); // Output: true
```

## **Explanation:**

1. **Serialization:**
  - o Perform preorder traversal to capture the structure of the tree.
  - o Perform inorder traversal to validate node order.
2. **Substring Check:**
  - o Check if the preorder and inorder traversals of subRoot are substrings of the corresponding traversals of root.
3. **Advantages:**
  - o Preorder ensures structure is preserved.
  - o Inorder ensures correct node ordering.
4. **Edge Cases:**
  - o Empty trees: Handle null trees gracefully.
  - o Large trees: Efficient due to substring checks.
5. **Time Complexity:**
  - o Serialization:  $O(n)$  for root and  $O(m)$  for subRoot.
  - o Substring Check:  $O(n + m)$ .
  - o Total:  $O(n + m)$ .
6. **Space Complexity:**
  - o  $O(n + m)$  for the serialized strings.

## **Checking Subtree Presence (Using Tree Hashing for Optimization)**

**280. Problem:** Optimize subtree detection using **hashing** to avoid redundant comparisons.

### **Solution:**

```
function isSubtreeUsingHash(root, subRoot) {  
    const hashSet = new Set();  
  
    function getTreeHash(node) {  
        if (!node) return "null";  
  
        const leftHash = getTreeHash(node.left);  
        const rightHash = getTreeHash(node.right);  
        const currentHash = `${node.val},${leftHash},${rightHash}`;  
  
        hashSet.add(currentHash);  
        return currentHash;  
    }  
  
    const subRootHash = getTreeHash(subRoot);  
    getTreeHash(root);  
  
    return hashSet.has(subRootHash);  
}  
  
// Test  
const root = new TreeNode(  
    3,  
    null, null
```

```

new TreeNode(4, new TreeNode(1), new TreeNode(2)),
new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeUsingHash(root, subRoot)); // Output: true

```

### **Explanation:**

1. **Hash Generation:**
  - o Compute a unique hash for each subtree by combining the node's value and the hashes of its children.
2. **Subtree Check:**
  - o Generate the hash for subRoot and check if it exists in the hash set of root.
3. **Advantages:**
  - o Efficient for trees with repeated structures.
  - o Avoids redundant comparisons.
4. **Time Complexity:**
  - o Hashing root:  $O(n)$ .
  - o Hashing subRoot:  $O(m)$ .
  - o Total:  $O(n + m)$ .
5. **Space Complexity:**
  - o  $O(n + m)$  for the hash set.

### **Checking Subtree Presence (Optimized Recursive with Depth Check)**

**281. Problem: Determine if subRoot is a subtree of root, optimizing by checking depth first to eliminate unnecessary comparisons.**

### **Solution:**

```

function isSubtreeWithDepth(root, subRoot) {
    function getDepth(node) {
        if (!node) return 0;
        return 1 + Math.max(getDepth(node.left), getDepth(node.right));
    }

    const rootDepth = getDepth(root);
    const subRootDepth = getDepth(subRoot);

    if (subRootDepth > rootDepth) return false;

    function isSameTree(tree1, tree2) {
        if (!tree1 && !tree2) return true;
        if (!tree1 || !tree2) return false;
        if (tree1.val !== tree2.val) return false;
    }
}

```

```

        return (
            isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right)
        );
    }

    function checkSubtree(node) {
        if (!node) return false;
        if (getDepth(node) < subRootDepth) return false; // Early exit
        if (isSameTree(node, subRoot)) return true;

        return checkSubtree(node.left) || checkSubtree(node.right);
    }

    return checkSubtree(root);
}

// Test
const root = new TreeNode(
    3,
    new TreeNode(4, new TreeNode(1), new TreeNode(2)),
    new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeWithDepth(root, subRoot)); // Output: true

```

### **Explanation:**

#### **1. Depth Calculation:**

- Use recursion to calculate the depth of both root and subRoot.
- If subRoot's depth exceeds root's depth, return false immediately.

#### **2. Recursive Subtree Check:**

- For each node in root, compare its depth with subRoot's depth.
- If subRoot's depth is greater at any point, skip further checks for that branch.

#### **3. Time Complexity:**

- Depth calculation:  $O(n)$  for root and  $O(m)$  for subRoot.
- Subtree check:  $O(m * n)$  in the worst case.

#### **4. Space Complexity:**

- $O(h)$  for the recursion stack, where  $h$  is the height of the tree.

### **Checking Subtree Presence (Top-Down Approach)**

**282. Problem: Determine if subRoot is a subtree of root using a simple top-down approach.**

### **Solution:**

```
function isSubtreeTopDown(root, subRoot) {
```

```

if (!root) return false; // Base case: Empty tree
if (isSameTree(root, subRoot)) return true;

// Recurse on left and right subtrees
return (
  isSubtreeTopDown(root.left, subRoot) ||
  isSubtreeTopDown(root.right, subRoot)
);
}

function isSameTree(tree1, tree2) {
  if (!tree1 && !tree2) return true;
  if (!tree1 || !tree2) return false;
  if (tree1.val !== tree2.val) return false;

  return (
    isSameTree(tree1.left, tree2.left) && isSameTree(tree1.right, tree2.right)
  );
}

// Test
const root = new TreeNode(
  3,
  new TreeNode(4, new TreeNode(1), new TreeNode(2)),
  new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreeTopDown(root, subRoot)); // Output: true

```

### **Explanation:**

#### **1. Recursive Traversal:**

- Start at the root of root and recursively traverse the left and right subtrees.
- At each node, check if it matches subRoot.

#### **2. Tree Comparison (isSameTree):**

- Compare two trees node by node.
- If all nodes match, the trees are identical.

#### **3. Time Complexity:**

- $O(m * n)$ , where m is the number of nodes in root and n is the number of nodes in subRoot.

#### **4. Space Complexity:**

- $O(h)$ , where h is the height of the tree.

### **Checking Subtree Presence (Using Postorder Traversal for Matching)**

**283. Problem: Determine if subRoot is a subtree of root using postorder traversal (left → right → root).**

### Solution:

```
function isSubtreePostorder(root, subRoot) {
    if (!root) return false; // Base case
    if (isSameTreePostorder(root, subRoot)) return true;

    // Check subtrees
    return (
        isSubtreePostorder(root.left, subRoot) ||
        isSubtreePostorder(root.right, subRoot)
    );
}

function isSameTreePostorder(tree1, tree2) {
    if (!tree1 && !tree2) return true; // Both are null
    if (!tree1 || !tree2) return false; // One is null
    if (tree1.val !== tree2.val) return false; // Values don't match

    // Compare left, right, and then root
    return (
        isSameTreePostorder(tree1.left, tree2.left) &&
        isSameTreePostorder(tree1.right, tree2.right)
    );
}

// Test
const root = new TreeNode(
    3,
    new TreeNode(4, new TreeNode(1), new TreeNode(2)),
    new TreeNode(5)
);
const subRoot = new TreeNode(4, new TreeNode(1), new TreeNode(2));

console.log(isSubtreePostorder(root, subRoot)); // Output: true
```

### Explanation:

#### 1. Postorder Traversal:

- o Compare left and right subtrees first, then the root.
- o Ensures the entire subtree matches before confirming a match.

#### 2. Tree Comparison:

- o Recursively check if two trees are identical using postorder traversal.

#### 3. Time Complexity:

- o  $O(m * n)$ .

#### 4. Space Complexity:

- o  $O(h)$ , where  $h$  is the height of the tree.

## Constructing Trees from Order Sequences

### 284. Problem:

Given two integer arrays preorder and inorder, where:

- preorder is the preorder traversal of a binary tree.
- inorder is the inorder traversal of the Comparing Two Tree Structures.

Construct and return the binary tree.

Example:

Input: preorder = [3, 9, 20, 15, 7], inorder = [9, 3, 15, 20, 7]

Output: Binary tree represented as:



Solution:

```
class TreeNode {
  constructor(val, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

function buildTree(preorder, inorder) {
  if (!preorder.length || !inorder.length) return null;

  const rootVal = preorder[0]; // The root is always the first element of preorder
  const root = new TreeNode(rootVal);

  const rootIndex = inorder.indexOf(rootVal); // Find the root in inorder traversal
  const leftInorder = inorder.slice(0, rootIndex); // Left subtree in inorder
  const rightInorder = inorder.slice(rootIndex + 1); // Right subtree in inorder

  const leftPreorder = preorder.slice(1, leftInorder.length + 1); // Left subtree in preorder
  const rightPreorder = preorder.slice(leftInorder.length + 1); // Right subtree in preorder

  root.left = buildTree(leftPreorder, leftInorder); // Recursively build left subtree
  root.right = buildTree(rightPreorder, rightInorder); // Recursively build right subtree

  return root;
}
```

```
// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTree(preorder, inorder);

console.log(tree);
```

### Explanation:

#### 1. Preorder Traversal Insight:

- o The first element in preorder is the root of the tree.

#### 2. Inorder Traversal Insight:

- o The root divides the inorder array into left and right subtrees:
  - Elements to the left of the root are in the left subtree.
  - Elements to the right of the root are in the right subtree.

#### 3. Recursive Steps:

- o Recursively divide the preorder and inorder arrays based on the root's index.
- o Rebuild the left and right subtrees using the divided arrays.

#### 4. Base Case:

- o If preorder or inorder is empty, return null.

#### 5. Time Complexity:

- o  $O(n^2)$ : For each root, we search for its index in the inorder array ( $O(n)$  for each recursive call).

#### 6. Space Complexity:

- o  $O(n)$ : Space required for the recursion stack in the worst case (skewed tree).

### Constructing Trees from Order Sequences (Optimized Using Hash Map)

**285. Problem: Given two integer arrays preorder and inorder, construct the binary tree more efficiently by avoiding repeated searches in the inorder array.**

### Solution:

```
class TreeNode {
  constructor(val, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

function buildTree(preorder, inorder) {
  const inorderIndexMap = new Map();
  inorder.forEach((val, idx) => inorderIndexMap.set(val, idx));

  function helper(preStart, inStart, inEnd) {
    if (preStart > preorder.length - 1 || inStart > inEnd) return null;
```

```

const rootVal = preorder[preStart];
const root = new TreeNode(rootVal);

const rootIndex = inorderIndexMap.get(rootVal);
const leftTreeSize = rootIndex - inStart;

root.left = helper(preStart + 1, inStart, rootIndex - 1);
root.right = helper(preStart + leftTreeSize + 1, rootIndex + 1, inEnd);

return root;
}

return helper(0, 0, inorder.length - 1);
}

// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTree(preorder, inorder);

console.log(tree);

```

### **Explanation:**

1. **Hash Map Optimization:**
  - o Build a hash map of the inorder array to store the indices of each value for O(1) lookups.
2. **Recursive Helper Function:**
  - o Use preStart to track the current root in the preorder array.
  - o Use inStart and inEnd to define the boundaries of the current subtree in the inorder array.
3. **Steps:**
  - o Identify the root node using preorder[preStart].
  - o Find the root's index in the inorder array using the hash map.
  - o Calculate the size of the left subtree (rootIndex - inStart).
  - o Recursively build the left and right subtrees.
4. **Base Case:**
  - o If preStart exceeds the length of preorder or inStart > inEnd, return null.
5. **Time Complexity:**
  - o O(n): Each node is processed once, and the hash map lookup is O(1).
6. **Space Complexity:**
  - o O(n): Space for the hash map and recursion stack.

### **Constructing Trees from Order Sequences (Iterative Approach)**

**286. Problem: Construct the binary tree using an iterative approach to avoid recursion.**

### **Solution:**

```

function buildTreeIterative(preorder, inorder) {
    if (!preorder.length || !inorder.length) return null;

    const root = new TreeNode(preorder[0]);
    const stack = [root];
    let inorderIndex = 0;

    for (let i = 1; i < preorder.length; i++) {
        const node = new TreeNode(preorder[i]);
        let parent = stack[stack.length - 1];

        if (parent.val !== inorder[inorderIndex]) {
            parent.left = node;
        } else {
            while (
                stack.length > 0 &&
                stack[stack.length - 1].val === inorder[inorderIndex]
            ) {
                parent = stack.pop();
                inorderIndex++;
            }
            parent.right = node;
        }

        stack.push(node);
    }

    return root;
}

// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTreeIterative(preorder, inorder);

console.log(tree);

```

## Explanation:

1. **Stack-Based Traversal:**
  - o Use a stack to keep track of nodes that have not yet been assigned a right child.
2. **Pointer in Inorder:**
  - o Maintain a pointer in the inorder array to track the next node to be processed.
3. **Steps:**
  - o Start with the first node in preorder as the root.
  - o For each subsequent node in preorder:

- If the current node is not the same as the current inorder node, it is the left child of the last node in the stack.
- Otherwise, pop nodes from the stack until the last popped node matches the current inorder node. The next node is the right child of the last popped node.

**4. Time Complexity:**

- $O(n)$ : Each node is processed once.

**5. Space Complexity:**

- $O(n)$ : Space for the stack.

### Constructing Trees from Order Sequences (Bottom-Up Recursive Approach)

**287. Problem: Construct a binary tree from preorder and inorder traversal using a bottom-up recursive approach.**

**Solution:**

```
function buildTreeBottomUp(preorder, inorder) {
  let preorderIndex = 0;

  const inorderIndexMap = new Map();
  inorder.forEach((val, idx) => inorderIndexMap.set(val, idx));

  function helper(inStart, inEnd) {
    if (inStart > inEnd) return null;

    const rootVal = preorder[preorderIndex++];
    const root = new TreeNode(rootVal);

    const rootIndex = inorderIndexMap.get(rootVal);

    root.left = helper(inStart, rootIndex - 1); // Build the left subtree
    root.right = helper(rootIndex + 1, inEnd); // Build the right subtree

    return root;
  }

  return helper(0, inorder.length - 1);
}

// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTreeBottomUp(preorder, inorder);

console.log(tree);
```

**Explanation:**

1. **Use a Pointer Instead of Slicing:**
  - o Maintain a preorderIndex to track the current root in the preorder array.
  - o Avoid slicing preorder and inorder arrays by using indices.
2. **Hash Map for Index Lookup:**
  - o Use a hash map to store indices of elements in the inorder array for O(1) lookups.
3. **Recursive Construction:**
  - o Define a recursive helper function that takes inStart and inEnd as arguments to define the current subtree range in the inorder array.
  - o Build the left and right subtrees recursively using the hash map to find the root's index in the inorder array.
4. **Base Case:**
  - o If inStart > inEnd, return null.
5. **Time Complexity:**
  - o O(n): Each node is processed once, and hash map lookups are O(1).
6. **Space Complexity:**
  - o O(n): Space for the hash map and recursion stack.

### **Constructing Trees from Order Sequences (With Boundary Validation)**

**288. Problem: Construct a binary tree and validate its boundaries during the recursive construction process.**

**Solution:**

```
function buildTreeWithBoundary(preorder, inorder) {
  let preorderIndex = 0;

  const inorderIndexMap = new Map();
  inorder.forEach((val, idx) => inorderIndexMap.set(val, idx));

  function helper(inStart, inEnd) {
    if (inStart > inEnd) return null;

    const rootVal = preorder[preorderIndex++];
    const root = new TreeNode(rootVal);

    const rootIndex = inorderIndexMap.get(rootVal);

    if (rootIndex < inStart || rootIndex > inEnd) {
      throw new Error("Invalid tree construction: Boundaries violated.");
    }

    root.left = helper(inStart, rootIndex - 1); // Build left subtree
    root.right = helper(rootIndex + 1, inEnd); // Build right subtree

    return root;
  }

  return helper(0, inorder.length - 1);
}
```

```

}

// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTreeWithBoundary(preorder, inorder);

console.log(tree);

```

### **Explanation:**

#### **1. Boundary Validation:**

- For each root, ensure its index in the inorder array lies within the range [inStart, inEnd].
- If the boundary is violated, throw an error.

#### **2. Recursive Construction:**

- Build the tree recursively while validating boundaries.

#### **3. Base Case:**

- If inStart > inEnd, return null.

#### **4. Time Complexity:**

- O(n): Each node is processed once, and hash map lookups are O(1).

#### **5. Space Complexity:**

- O(n): Space for the hash map and recursion stack.

### **Constructing Trees from Order Sequences (Compact Iterative Approach)**

**289. Problem: Construct the binary tree iteratively in a compact manner without explicitly maintaining left and right subtrees in the stack.**

### **Solution:**

```

function buildTreeCompactIterative(preorder, inorder) {
  if (!preorder.length || !inorder.length) return null;

  const root = new TreeNode(preorder[0]);
  const stack = [root];
  let inorderIndex = 0;

  for (let i = 1; i < preorder.length; i++) {
    const node = new TreeNode(preorder[i]);
    let parent = stack[stack.length - 1];

    if (parent.val !== inorder[inorderIndex]) {
      parent.left = node;
    } else {
      while (
        stack.length > 0 &&
        stack[stack.length - 1].val === inorder[inorderIndex]
      ) {
        parent.right = node;
        parent = stack.pop();
        inorderIndex++;
      }
      parent.right = node;
    }
    stack.push(node);
  }
}

```

```

        ) {
            parent = stack.pop();
            inorderIndex++;
        }
        parent.right = node;
    }

    stack.push(node);
}

return root;
}

// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTreeCompactIterative(preorder, inorder);

console.log(tree);

```

### Explanation:

1. **Single Stack:**
  - o Maintain a stack to track nodes whose right child hasn't been assigned yet.
2. **Pointer in inorder:**
  - o Use a pointer in the inorder array to decide when to move to the right subtree.
3. **Steps:**
  - o If the top of the stack doesn't match the current inorder element, the new node is a left child.
  - o Otherwise, pop from the stack until the top matches the inorder element and assign the new node as the right child.
4. **Time Complexity:**
  - o  $O(n)$ : Each node is processed once.
5. **Space Complexity:**
  - o  $O(h)$ , where  $h$  is the height of the tree.

### Constructing Trees from Order Sequences (Handling Edge Cases)

#### 290. Problem:

Ensure the tree can be constructed properly for edge cases, such as:

1. Trees with only left or right children.
2. Completely unbalanced trees (skewed left or right).
3. Single-node trees or empty trees.

### Solution:

```

function buildTreeEdgeCases(preorder, inorder) {
    if (!preorder.length || !inorder.length) return null;
}

```

```

let preorderIndex = 0;
const inorderIndexMap = new Map();
inorder.forEach((val, idx) => inorderIndexMap.set(val, idx));

function helper(inStart, inEnd) {
  if (inStart > inEnd) return null;

  const rootVal = preorder[preorderIndex++];
  const root = new TreeNode(rootVal);

  const rootIndex = inorderIndexMap.get(rootVal);

  root.left = helper(inStart, rootIndex - 1); // Build left subtree
  root.right = helper(rootIndex + 1, inEnd); // Build right subtree

  return root;
}

return helper(0, inorder.length - 1);
}

// Test: Skewed Left Tree
const preorderLeft = [3, 2, 1];
const inorderLeft = [1, 2, 3];
const skewedLeftTree = buildTreeEdgeCases(preorderLeft, inorderLeft);
console.log(skewedLeftTree);

// Test: Skewed Right Tree
const preorderRight = [1, 2, 3];
const inorderRight = [1, 2, 3];
const skewedRightTree = buildTreeEdgeCases(preorderRight, inorderRight);
console.log(skewedRightTree);

// Test: Single Node Tree
const preorderSingle = [1];
const inorderSingle = [1];
const singleNodeTree = buildTreeEdgeCases(preorderSingle, inorderSingle);
console.log(singleNodeTree);

// Test: Empty Tree
const preorderEmpty = [];
const inorderEmpty = [];
const emptyTree = buildTreeEdgeCases(preorderEmpty, inorderEmpty);
console.log(emptyTree);

```

### **Explanation:**

- 1. Edge Case Handling:**

- Empty tree: Return null if preorder or inorder is empty.
  - Skewed trees: Ensure the recursive logic handles cases where one subtree is always empty.
  - Single-node trees: Ensure the base case handles trees with one node correctly.
2. **Steps:**
    - Validate input arrays before proceeding with tree construction.
    - Use the recursive helper function to construct subtrees based on inorder boundaries.
  3. **Base Case:**
    - If  $\text{inStart} > \text{inEnd}$ , return null.
  4. **Time Complexity:**
    - $O(n)$ : Each node is processed once, and hash map lookups are  $O(1)$ .
  5. **Space Complexity:**
    - $O(n)$ : Space for the hash map and recursion stack.

### **Constructing Trees from Order Sequences (Compact Solution with Index Tracking)**

**291. Problem: Simplify the implementation by avoiding unnecessary variable declarations and focusing on compactness.**

**Solution:**

```
function buildTreeCompact(preorder, inorder) {
  const inorderMap = new Map();
  inorder.forEach((val, idx) => inorderMap.set(val, idx));

  let preorderIndex = 0;

  function build(inStart, inEnd) {
    if (inStart > inEnd) return null;

    const root = new TreeNode(preorder[preorderIndex++]);
    const rootIndex = inorderMap.get(root.val);

    root.left = build(inStart, rootIndex - 1);
    root.right = build(rootIndex + 1, inEnd);

    return root;
  }

  return build(0, inorder.length - 1);
}

// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTreeCompact(preorder, inorder);
console.log(tree);
```

## Explanation:

### 1. Compact Design:

- Avoid unnecessary variables for left and right subtrees.
- Use concise logic to define boundaries and recurse.

### 2. Efficiency:

- Same time complexity:  $O(n)$ .
- Reduced code length improves readability.

## Constructing Trees from Order Sequences (Iterative with Optimized Space)

**292. Problem: Construct the binary tree iteratively while minimizing space usage.**

## Solution:

```
function buildTreeIterativeOptimized(preorder, inorder) {  
    if (!preorder.length || !inorder.length) return null;  
  
    const root = new TreeNode(preorder[0]);  
    const stack = [root];  
    let inorderIndex = 0;  
  
    for (let i = 1; i < preorder.length; i++) {  
        const node = new TreeNode(preorder[i]);  
        let parent = stack[stack.length - 1];  
  
        if (parent.val !== inorder[inorderIndex]) {  
            parent.left = node;  
        } else {  
            while (  
                stack.length &&  
                stack[stack.length - 1].val === inorder[inorderIndex]  
            ) {  
                parent = stack.pop();  
                inorderIndex++;  
            }  
            parent.right = node;  
        }  
  
        stack.push(node);  
    }  
  
    return root;  
}  
  
// Test  
const preorder = [3, 9, 20, 15, 7];  
const inorder = [9, 3, 15, 20, 7];  
const tree = buildTreeIterativeOptimized(preorder, inorder);
```

```
console.log(tree);
```

### Explanation:

#### 1. Stack Management:

- o Use a single stack to keep track of nodes whose children are yet to be assigned.
- o Avoid redundant stack space usage by reusing the same stack.

#### 2. Pointer in Inorder:

- o Track the current position in the inorder array to determine when to move to the right subtree.

#### 3. Time Complexity:

- o  $O(n)$ : Each node is processed once.

#### 4. Space Complexity:

- o  $O(h)$ , where  $h$  is the height of the tree.

## Constructing Trees from Order Sequences (Using a Deque for Efficient Traversal)

### 293. Problem: Construct a binary tree from preorder and inorder traversals, using a deque to optimize operations on both ends.

#### Solution:

```
function buildTreeWithDequeue(preorder, inorder) {  
    if (!preorder.length || !inorder.length) return null;  
  
    const inorderIndexMap = new Map();  
    inorder.forEach((val, idx) => inorderIndexMap.set(val, idx));  
  
    const deque = preorder.slice(); // Initialize deque with preorder values  
  
    function buildTree(inStart, inEnd) {  
        if (inStart > inEnd) return null;  
  
        const rootVal = deque.shift(); // Remove the front element  
        const root = new TreeNode(rootVal);  
  
        const rootIndex = inorderIndexMap.get(rootVal);  
  
        root.left = buildTree(inStart, rootIndex - 1);  
        root.right = buildTree(rootIndex + 1, inEnd);  
  
        return root;  
    }  
  
    return buildTree(0, inorder.length - 1);  
}
```

```
// Test
const preorder = [3, 9, 20, 15, 7];
const inorder = [9, 3, 15, 20, 7];
const tree = buildTreeWithDequeue(preorder, inorder);

console.log(tree);
```

### **Explanation:**

#### **1. Deque for Preorder:**

- Use a deque (or shift operation on an array) to efficiently manage the preorder traversal. The root of the current subtree is always at the front of the deque.

#### **2. Steps:**

- Build the hash map for quick lookup of indices in inorder.
- Recursively construct left and right subtrees using inStart and inEnd indices.

#### **3. Advantages:**

- Efficient front removal with a deque ensures smooth traversal of preorder.

#### **4. Time Complexity:**

- $O(n)$ : Each node is processed once.

#### **5. Space Complexity:**

- $O(n)$ : Space for the deque and recursion stack.

## **Validate Binary Search Tree (Recursive Approach)**

### **294. Problem:**

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A binary search tree (BST) is valid if:

1. The left subtree of a node contains only nodes with values less than the node's value.
2. The right subtree of a node contains only nodes with values greater than the node's value.
3. Both the left and right subtrees must also be binary search trees.

### **Example:**

Input: root = [2, 1, 3]  
Output: true

Input: root = [5, 1, 4, null, null, 3, 6]  
Output: false

Explanation: The root node's value is 5 but its right child contains a node with value 4, which violates the BST rules.

### **Solution:**

```
function isValidBST(root) {
    function validate(node, min, max) {
```

```

if (!node) return true; // An empty tree is valid

if (
  (min !== null && node.val <= min) ||
  (max !== null && node.val >= max)
) {
  return false; // Current node violates the min/max constraint
}

// Recursively validate the left and right subtrees
return (
  validate(node.left, min, node.val) && validate(node.right, node.val, max)
);
}

return validate(root, null, null); // Initially, there are no constraints
}

// Test
class TreeNode {
  constructor(val, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBST(root1)); // Output: true

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBST(root2)); // Output: false

```

## Explanation:

### 1. Constraints:

- o Use min and max to define the range of valid values for the current node.
- o If node.val is outside this range, the tree is invalid.

### 2. Recursive Steps:

- o For the left subtree, update max to the current node's value.
- o For the right subtree, update min to the current node's value.

### 3. Base Case:

- o If the node is null, return true because an empty subtree is valid.

### 4. Time Complexity:

- $O(n)$ , where  $n$  is the number of nodes in the tree. Each node is visited once.
5. **Space Complexity:**
- $O(h)$ , where  $h$  is the height of the tree (due to the recursion stack).

### Validate Binary Search Tree (Iterative Approach)

**295. Problem:** Validate a binary search tree (BST) using an **iterative approach**. The iterative method uses a stack to simulate the recursive behavior while validating the BST rules.

**Solution:**

```
function isValidBSTIterative(root) {
    if (!root) return true;

    const stack = [];
    let prev = null; // Track the previous node's value during traversal

    while (stack.length > 0 || root) {
        while (root) {
            stack.push(root);
            root = root.left; // Traverse left subtree
        }

        root = stack.pop(); // Visit the node

        // If the current node's value is not greater than the previous, it's invalid
        if (prev !== null && root.val <= prev) {
            return false;
        }

        prev = root.val; // Update the previous node value
        root = root.right; // Traverse right subtree
    }

    return true;
}

// Test
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBSTIterative(root1)); // Output: true

const root2 = new TreeNode(
    5,
    new TreeNode(1),
    new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTIterative(root2)); // Output: false
```

## **Explanation:**

### **1. Inorder Traversal:**

- A valid BST will produce a strictly increasing sequence when traversed in inorder (left → root → right).
- Use a stack to traverse the tree iteratively while maintaining the order.

### **2. Validation:**

- Keep track of the previous node's value (prev).
- Ensure the current node's value is strictly greater than prev.

### **3. Steps:**

- Push all left nodes onto the stack until a leaf is reached.
- Pop a node from the stack, validate it, and traverse its right subtree.

### **4. Edge Cases:**

- Empty tree: Return true.
- Single-node tree: Return true.

### **5. Time Complexity:**

- $O(n)$ : Each node is visited once.

### **6. Space Complexity:**

- $O(h)$ : Space for the stack, where  $h$  is the height of the tree.

## **Validate Binary Search Tree (Using Inorder Traversal and Array)**

**296. Problem: Store the inorder traversal of the tree in an array and check if it is strictly increasing.**

### **Solution:**

```
function isValidBSTInorderArray(root) {  
    const inorder = [];  
  
    function inorderTraversal(node) {  
        if (!node) return;  
        inorderTraversal(node.left);  
        inorder.push(node.val); // Store the node's value  
        inorderTraversal(node.right);  
    }  
  
    inorderTraversal(root);  
  
    // Check if the array is sorted in strictly increasing order  
    for (let i = 1; i < inorder.length; i++) {  
        if (inorder[i] <= inorder[i - 1]) {  
            return false;  
        }  
    }  
  
    return true;  
}  
  
// Test
```

```

const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBSTInorderArray(root1)); // Output: true

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTInorderArray(root2)); // Output: false

```

### **Explanation:**

#### **1. Inorder Traversal:**

- Perform a recursive inorder traversal of the tree.
- Store the values in an array.

#### **2. Validation:**

- Check if the array is sorted in strictly increasing order.
- If any two consecutive elements are not in increasing order, the tree is invalid.

#### **3. Edge Cases:**

- Empty tree: Return true.
- Single-node tree: Return true.

#### **4. Time Complexity:**

- $O(n)$ : Inorder traversal visits each node once.

#### **5. Space Complexity:**

- $O(n)$ : Space for the inorder array.

### **Validate Binary Search Tree (Optimized Recursive with Early Exit)**

**297. Problem: Optimize the recursive approach by implementing early exit to avoid unnecessary traversals.**

### **Solution:**

```

function isValidBSTOptimized(root) {
  function validate(node, min, max) {
    if (!node) return true;

    if (
      (min !== null && node.val <= min) ||
      (max !== null && node.val >= max)
    ) {
      return false; // Early exit if constraints are violated
    }

    // Recursively validate left and right subtrees
    return (
      validate(node.left, min, node.val) && validate(node.right, node.val, max)
    );
  }
}

```

```

    }

    return validate(root, null, null);
}

// Test
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBSTOptimized(root1)); // Output: true

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTOptimized(root2)); // Output: false

```

### Explanation:

1. **Validation:**
  - o Similar to the recursive approach, but with an early exit to stop further recursion as soon as a violation is detected.
2. **Early Exit:**
  - o If a node violates the BST constraints, return false immediately without checking further.
3. **Edge Cases:**
  - o Empty tree: Return true.
  - o Single-node tree: Return true.
4. **Time Complexity:**
  - o  $O(n)$ : Each node is visited once.
5. **Space Complexity:**
  - o  $O(h)$ : Space for the recursion stack.

### Validate Binary Search Tree (Bottom-Up Recursive Approach)

**298. Problem: Validate a binary search tree (BST) using a bottom-up recursive approach. This approach validates the left and right subtrees first and passes the valid range up the recursion stack.**

### Solution:

```

function isValidBSTBottomUp(root) {
  function validate(node) {
    if (!node) return [true, null, null]; // Base case: [isValid, minValue, maxValue]

    const [leftValid, leftMin, leftMax] = validate(node.left);
    const [rightValid, rightMin, rightMax] = validate(node.right);

    const isValid =
      leftValid && rightValid &&
      node.value > leftMax && node.value < rightMin;
    const minValue = Math.min(leftMin, rightMin);
    const maxValue = Math.max(leftMax, rightMax);

    return [isValid, minValue, maxValue];
  }

  validate(root);
}

```

```

if (!leftValid || !rightValid) return [false, null, null];
if (
  (leftMax !== null && node.val <= leftMax) ||
  (rightMin !== null && node.val >= rightMin)
) {
  return [false, null, null];
}

// Update the min and max values for the current subtree
const minValue = leftMin !== null ? leftMin : node.val;
const maxValue = rightMax !== null ? rightMax : node.val;

return [true, minValue, maxValue];
}

return validate(root)[0];
}

// Test
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBSTBottomUp(root1)); // Output: true

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTBottomUp(root2)); // Output: false

```

### Explanation:

#### 1. Return Values:

- Each recursive call returns [isValid, minValue, maxValue] for the current subtree.
- isValid: Whether the subtree is a valid BST.
- minValue and maxValue: The minimum and maximum values in the current subtree.

#### 2. Validation Logic:

- Check if the current node's value satisfies the constraints imposed by its left and right subtrees.
- If valid, update the minValue and maxValue for the current subtree.

#### 3. Base Case:

- If the node is null, return [true, null, null].

#### 4. Time Complexity:

- O(n): Each node is visited once.

#### 5. Space Complexity:

- O(h): Space for the recursion stack, where h is the height of the tree.

## Validate Binary Search Tree (Using Morris Traversal for Inorder Validation)

**299. Problem:** Validate a binary search tree (BST) using Morris Traversal, which performs an inorder traversal with O(1) space.

**Solution:**

```
function isValidBSTMorris(root) {  
    let prev = null;  
  
    while (root) {  
        if (!root.left) {  
            if (prev !== null && root.val <= prev) return false;  
            prev = root.val;  
            root = root.right;  
        } else {  
            let predecessor = root.left;  
  
            // Find the rightmost node in the left subtree  
            while (predecessor.right && predecessor.right !== root) {  
                predecessor = predecessor.right;  
            }  
  
            if (!predecessor.right) {  
                predecessor.right = root; // Create a temporary link  
                root = root.left;  
            } else {  
                predecessor.right = null; // Remove the temporary link  
                if (prev !== null && root.val <= prev) return false;  
                prev = root.val;  
                root = root.right;  
            }  
        }  
    }  
  
    return true;  
}  
  
// Test  
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));  
console.log(isValidBSTMorris(root1)); // Output: true  
  
const root2 = new TreeNode(  
    5,  
    new TreeNode(1),  
    new TreeNode(4, new TreeNode(3), new TreeNode(6))  
);  
console.log(isValidBSTMorris(root2)); // Output: false
```

## Explanation:

### 1. Morris Traversal:

- o Uses temporary links in the tree to traverse without recursion or a stack.
- o Performs an inorder traversal (left → root → right).

### 2. Validation:

- o During the traversal, check if the current node's value is greater than the previous node's value.

### 3. Edge Cases:

- o Empty tree: Return true.
- o Single-node tree: Return true.

### 4. Time Complexity:

- o  $O(n)$ : Each node is visited at most twice.

### 5. Space Complexity:

- o  $O(1)$ : No additional space is used, aside from variables.

## Validate Binary Search Tree (Using Level-Order Traversal with Ranges)

**300. Problem: Validate a binary search tree (BST) using a level-order traversal (BFS), with min and max ranges for each node.**

## Solution:

```
function isValidBSTLevelOrder(root) {  
    if (!root) return true;  
  
    const queue = [[root, null, null]]; // [node, min, max]  
  
    while (queue.length > 0) {  
        const [node, min, max] = queue.shift();  
  
        if (  
            (min !== null && node.val <= min) ||  
            (max !== null && node.val >= max)  
        ) {  
            return false;  
        }  
  
        if (node.left) queue.push([node.left, min, node.val]);  
        if (node.right) queue.push([node.right, node.val, max]);  
    }  
  
    return true;  
}  
  
// Test  
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));  
console.log(isValidBSTLevelOrder(root1)); // Output: true
```

```

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTLevelOrder(root2)); // Output: false

```

### **Explanation:**

#### **1. Level-Order Traversal:**

- Use a queue to process nodes level by level.

#### **2. Validation:**

- Check if the current node's value falls within the valid range defined by its ancestors.

#### **3. Time Complexity:**

- $O(n)$ : Each node is processed once.

#### **4. Space Complexity:**

- $O(n)$ : Space for the queue.

### **Validate Binary Search Tree (Using Binary Search in Subtree Validation)**

#### **301. Problem: Validate a binary search tree (BST) by directly applying the binary search principle on subtrees.**

### **Solution:**

```

function isValidBSTBinarySearch(root) {
  function isBST(node, min, max) {
    if (!node) return true;

    if (
      (min !== null && node.val <= min) ||
      (max !== null && node.val >= max)
    ) {
      return false; // Node violates the BST condition
    }

    // Validate left and right subtrees
    return isBST(node.left, min, node.val) && isBST(node.right, node.val, max);
  }

  return isBST(root, null, null);
}

// Test
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBSTBinarySearch(root1)); // Output: true

```

```

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTBinarySearch(root2)); // Output: false

```

### **Explanation:**

#### **1. Binary Search Principle:**

- o For the left subtree, the maximum value is the current node's value.
- o For the right subtree, the minimum value is the current node's value.

#### **2. Recursive Validation:**

- o Pass the updated min and max constraints recursively to child nodes.

#### **3. Base Case:**

- o If the node is null, return true (an empty subtree is valid).

#### **4. Time Complexity:**

- o  $O(n)$ : Each node is visited once.

#### **5. Space Complexity:**

- o  $O(h)$ : Space for the recursion stack, where  $h$  is the height of the tree.

### **Validate Binary Search Tree (Using Preorder Traversal)**

**302. Problem:** Validate a BST by traversing the tree in **preorder** ( $\text{root} \rightarrow \text{left} \rightarrow \text{right}$ ) and maintaining constraints for each node.

### **Solution:**

```

function isValidBSTPreorder(root) {
  function validate(node, min, max) {
    if (!node) return true;

    if (
      (min !== null && node.val <= min) ||
      (max !== null && node.val >= max)
    ) {
      return false; // Node violates the BST condition
    }

    // Validate left and right subtrees recursively
    return (
      validate(node.left, min, node.val) && validate(node.right, node.val, max)
    );
  }

  return validate(root, null, null);
}

```

```
// Test
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBSTPreorder(root1)); // Output: true

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTPreorder(root2)); // Output: false
```

### Explanation:

#### 1. Preorder Traversal:

- o Validate the current node first, then recursively validate the left and right subtrees.

#### 2. Validation Constraints:

- o Pass min and max constraints to ensure that all nodes adhere to the BST rules.

#### 3. Base Case:

- o If the node is null, return true.

#### 4. Time Complexity:

- o  $O(n)$ : Each node is visited once.

#### 5. Space Complexity:

- o  $O(h)$ : Space for the recursion stack.

### Validate Binary Search Tree (Combination of Inorder and Recursive Validation)

**303. Problem: Combine the principles of recursive validation and inorder traversal to ensure strict BST validation.**

### Solution:

```
function isValidBSTCombined(root) {
  let prev = null;

  function validate(node) {
    if (!node) return true;

    // Check left subtree
    if (!validate(node.left)) return false;

    // Inorder check
    if (prev !== null && node.val <= prev) return false;

    prev = node.val;

    // Check right subtree
    return validate(node.right);
  }

  validate(root);
}
```

```

    }

    return validate(root);
}

// Test
const root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
console.log(isValidBSTCombined(root1)); // Output: true

const root2 = new TreeNode(
  5,
  new TreeNode(1),
  new TreeNode(4, new TreeNode(3), new TreeNode(6))
);
console.log(isValidBSTCombined(root2)); // Output: false

```

### Explanation:

**1. Inorder Validation:**

- o Use prev to keep track of the previous node during inorder traversal.
- o Ensure that the current node's value is greater than prev.

**2. Recursive Validation:**

- o Validate the left and right subtrees recursively.

**3. Base Case:**

- o If the node is null, return true.

**4. Time Complexity:**

- o  $O(n)$ : Each node is visited once.

**5. Space Complexity:**

- o  $O(h)$ : Space for the recursion stack.

### Kth Smallest Element in a BST (Recursive Inorder Traversal)

**304. Problem:** Given the root of a binary search tree (BST) and an integer k, return the **kth smallest element** in the BST.

A BST's inorder traversal produces nodes in ascending order.

### Solution:

```

function kthSmallest(root, k) {
  let count = 0; // Keep track of the number of nodes visited
  let result = null; // Store the kth smallest value

  function inorder(node) {
    if (!node || result !== null) return;

    inorder(node.left); // Visit left subtree

```

```

count++;
if (count === k) {
    result = node.val; // Found the kth smallest
    return;
}

inorder(node.right); // Visit right subtree
}

inorder(root);
return result;
}

// Test
class TreeNode {
    constructor(val, left = null, right = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

const root = new TreeNode(
    3,
    new TreeNode(1, null, new TreeNode(2)),
    new TreeNode(4)
);
console.log(kthSmallest(root, 1)); // Output: 1
console.log(kthSmallest(root, 3)); // Output: 3

```

## Explanation:

### 1. Inorder Traversal:

- o Perform an inorder traversal (left → root → right).
- o Nodes are visited in ascending order of their values.

### 2. Counting Nodes:

- o Increment the counter each time a node is visited.
- o Stop traversal once the kth node is found.

### 3. Base Case:

- o Stop the recursion if the node is null or if the kth smallest element is already found.

### 4. Edge Cases:

- o k is larger than the number of nodes: Return null or throw an error.
- o root is null: Return null.

### 5. Time Complexity:

- o  $O(h + k)$ : Traverse up to k nodes and their ancestors, where h is the height of the tree.

### 6. Space Complexity:

- o  $O(h)$ : Space for the recursion stack.

### Kth Smallest Element in a BST (Iterative Inorder Traversal)

**305. Problem:** Find the kth smallest element in a BST using an iterative approach to perform an inorder traversal.

**Solution:**

```
function kthSmallestIterative(root, k) {
    const stack = [];
    let count = 0;

    while (stack.length > 0 || root) {
        // Traverse to the leftmost node
        while (root) {
            stack.push(root);
            root = root.left;
        }

        // Visit the node
        root = stack.pop();
        count++;

        if (count === k) {
            return root.val; // Found the kth smallest
        }

        // Move to the right subtree
        root = root.right;
    }

    return null; // If k is invalid
}

// Test
const root = new TreeNode(
    3,
    new TreeNode(1, null, new TreeNode(2)),
    new TreeNode(4)
);
console.log(kthSmallestIterative(root, 1)); // Output: 1
console.log(kthSmallestIterative(root, 3)); // Output: 3
```

**Explanation:**

**1. Inorder Traversal:**

- o A stack is used to traverse the tree iteratively.

- Left children are pushed onto the stack until a leaf is reached.
2. **Visit Node:**
    - Nodes are popped from the stack and visited in ascending order.
  3. **Counting Nodes:**
    - Increment the counter when visiting a node.
    - Stop once the kth node is reached.
  4. **Edge Cases:**
    - If k is greater than the total number of nodes, return null.
    - If the tree is empty (root is null), return null.
  5. **Time Complexity:**
    - $O(h + k)$ : Traverse up to k nodes and their ancestors, where h is the height of the tree.
  6. **Space Complexity:**
    - $O(h)$ : Space for the stack.

### **Kth Smallest Element in a BST (Optimized with Node Count)**

**306. Problem: Optimize the solution by maintaining the count of nodes in the left subtree for each node.**

**Solution:**

```
class TreeNodeWithCount {
    constructor(val, left = null, right = null, count = 1) {
        this.val = val;
        this.left = left;
        this.right = right;
        this.count = count; // Total nodes in the subtree rooted at this node
    }
}

function kthSmallestOptimized(root, k) {
    function getCount(node) {
        return node ? node.count : 0;
    }

    let current = root;

    while (current) {
        const leftCount = getCount(current.left);

        if (k === leftCount + 1) {
            return current.val; // Current node is the kth smallest
        } else if (k <= leftCount) {
            current = current.left; // Go to the left subtree
        } else {
            k -= leftCount + 1; // Exclude left subtree and current node
            current = current.right; // Go to the right subtree
        }
    }
}
```

```

    }

    return null; // If k is invalid
}

// Example usage
const root = new TreeNodeWithCount(
  3,
  new TreeNodeWithCount(1, null, new TreeNodeWithCount(2)),
  new TreeNodeWithCount(4)
);
console.log(kthSmallestOptimized(root, 1)); // Output: 1
console.log(kthSmallestOptimized(root, 3)); // Output: 3

```

### **Explanation:**

#### **1. Augmented Tree:**

- o Each node stores the total number of nodes in its subtree.

#### **2. Logic:**

- o Use the left subtree count to decide whether the kth smallest element is in the left subtree, the current node, or the right subtree.

#### **3. Edge Cases:**

- o Invalid k: Return null.
- o Empty tree: Return null.

#### **4. Time Complexity:**

- o  $O(h)$ : Traverse the height of the tree, where h is the height of the BST.

#### **5. Space Complexity:**

- o  $O(1)$ : No additional space is used apart from the input tree.

### **Kth Smallest Element in a BST (Morris Traversal)**

**307. Problem: Find the kth smallest element in a BST using Morris Traversal, which performs an inorder traversal with  $O(1)$  space.**

### **Solution:**

```

function kthSmallestMorris(root, k) {
  let count = 0;
  let result = null;

  while (root) {
    if (!root.left) {
      count++;
      if (count === k) {
        result = root.val;
        break;
      }
      root = root.right;
    } else {
      let pre = root.left;
      while (pre.right === root) {
        pre = pre.right;
      }
      if (pre.right === null) {
        pre.right = root;
        root = root.left;
      } else {
        pre.right = null;
        count++;
        if (count === k) {
          result = root.val;
          break;
        }
        root = root.right;
      }
    }
  }
  return result;
}

```

```

} else {
    let predecessor = root.left;

    while (predecessor.right && predecessor.right !== root) {
        predecessor = predecessor.right;
    }

    if (!predecessor.right) {
        predecessor.right = root;
        root = root.left;
    } else {
        predecessor.right = null;
        count++;
        if (count === k) {
            result = root.val;
            break;
        }
        root = root.right;
    }
}

return result;
}

// Test
const root = new TreeNode(
    3,
    new TreeNode(1, null, new TreeNode(2)),
    new TreeNode(4)
);
console.log(kthSmallestMorris(root, 1)); // Output: 1
console.log(kthSmallestMorris(root, 3)); // Output: 3

```

### Explanation:

#### 1. Morris Traversal:

- Temporarily modify the tree to perform an inorder traversal.
- Restore the tree structure after the traversal.

#### 2. Counting Nodes:

- Increment the counter during the traversal and stop when the kth node is reached.

#### 3. Edge Cases:

- Invalid k: Return null.
- Empty tree: Return null.

#### 4. Time Complexity:

- O(n): Each node is visited at most twice.

#### 5. Space Complexity:

- O(1): No additional space is used.

### **Kth Smallest Element in a BST (Using Binary Search on Inorder Traversal)**

**308. Problem:** **Find the kth smallest element in a BST by first performing an inorder traversal to create a sorted list of elements, then using binary search to access the kth element.**

**Solution:**

```
function kthSmallestBinarySearch(root, k) {
  const elements = [];

  function inorder(node) {
    if (!node) return;
    inorder(node.left);
    elements.push(node.val);
    inorder(node.right);
  }

  inorder(root);

  return elements[k - 1] || null; // Return the kth element (1-based indexing)
}

// Test
const root = new TreeNode(
  3,
  new TreeNode(1, null, new TreeNode(2)),
  new TreeNode(4)
);
console.log(kthSmallestBinarySearch(root, 1)); // Output: 1
console.log(kthSmallestBinarySearch(root, 3)); // Output: 3
```

**Explanation:**

- 1. Inorder Traversal:**
  - Extracts the elements of the BST in sorted order.
- 2. Binary Search:**
  - Access the k - 1th element in the sorted array directly.
- 3. Edge Cases:**
  - If k exceeds the total number of nodes, return null.
- 4. Time Complexity:**
  - O(n): Inorder traversal processes all nodes once.
- 5. Space Complexity:**
  - O(n): Space for the elements array.

### **Kth Smallest Element in a BST (Using Priority Queue/Min-Heap)**

**309. Problem:** Use a **min-heap** to dynamically track the smallest elements in the BST and extract the kth smallest.

**Solution:**

```
function kthSmallestMinHeap(root, k) {  
    const minHeap = [];  
  
    function inorder(node) {  
        if (!node) return;  
        inorder(node.left);  
        minHeap.push(node.val); // Push each element into the min-heap  
        inorder(node.right);  
    }  
  
    inorder(root);  
  
    return minHeap[k - 1] || null; // Extract the kth smallest element  
}  
  
// Test  
const root = new TreeNode(  
    3,  
    new TreeNode(1, null, new TreeNode(2)),  
    new TreeNode(4)  
);  
console.log(kthSmallestMinHeap(root, 1)); // Output: 1  
console.log(kthSmallestMinHeap(root, 3)); // Output: 3
```

**Explanation:**

1. **Min-Heap:**
  - o Store elements in sorted order (essentially acting as an array in this case).
2. **Edge Cases:**
  - o If k is larger than the number of nodes, return null.
3. **Time Complexity:**
  - o  $O(n)$ : Inorder traversal visits all nodes.
4. **Space Complexity:**
  - o  $O(n)$ : Space for the heap.

### Kth Smallest Element in a BST (Optimized for Constant Memory Access)

**310. Problem:** If the BST remains unchanged, perform a one-time traversal to store all elements in a sorted array. Future queries for any kth smallest element can be answered in  $O(1)$  time.

**Solution:**

```
class KthSmallestCache {
```

```

constructor(root) {
    this.elements = [];
    this.buildCache(root);
}

buildCache(node) {
    if (!node) return;
    this.buildCache(node.left);
    this.elements.push(node.val);
    this.buildCache(node.right);
}

kthSmallest(k) {
    return this.elements[k - 1] || null; // Access the kth smallest
}
}

// Test
const root = new TreeNode(
    3,
    new TreeNode(1, null, new TreeNode(2)),
    new TreeNode(4)
);
const cache = new KthSmallestCache(root);
console.log(cache.kthSmallest(1)); // Output: 1
console.log(cache.kthSmallest(3)); // Output: 3

```

### **Explanation:**

1. **Cache Construction:**
  - o Build a sorted array of all elements using inorder traversal.
2. **Query Time:**
  - o  $O(1)$  for each query to fetch the kth smallest element.
3. **Edge Cases:**
  - o If k exceeds the number of nodes, return null.
4. **Time Complexity:**
  - o  $O(n)$  for the initial traversal to build the cache.
  - o  $O(1)$  for each query.
5. **Space Complexity:**
  - o  $O(n)$ : Space for the elements array.

### **Lowest Common Ancestor of a BST (Recursive Approach)**

**311. Problem: Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes p and q.**

**The LCA of two nodes p and q in a BST is defined as the lowest node in the tree that has both p and q as descendants.**

**Example:**

Input: root = [6, 2, 8, 0, 4, 7, 9, null, null, 3, 5], p = 2, q = 8

Output: 6

Explanation: The root node (6) is the LCA of nodes 2 and 8.

Input: root = [6, 2, 8, 0, 4, 7, 9, null, null, 3, 5], p = 2, q = 4

Output: 2

Explanation: The node 2 is the LCA of nodes 2 and 4, as it is an ancestor of itself.

**Solution:**

```
function lowestCommonAncestor(root, p, q) {  
    if (!root) return null;  
  
    if (p.val < root.val && q.val < root.val) {  
        return lowestCommonAncestor(root.left, p, q); // LCA is in the left subtree  
    }  
  
    if (p.val > root.val && q.val > root.val) {  
        return lowestCommonAncestor(root.right, p, q); // LCA is in the right subtree  
    }  
  
    return root; // Current node is the LCA  
}  
  
// Test  
class TreeNode {  
    constructor(val, left = null, right = null) {  
        this.val = val;  
        this.left = left;  
        this.right = right;  
    }
}  
  
const root = new TreeNode(  
    6,  
    new TreeNode(  
        2,  
        new TreeNode(0),  
        new TreeNode(4, new TreeNode(3), new TreeNode(5))  
    ),  
    new TreeNode(8, new TreeNode(7), new TreeNode(9))  
);  
const p = new TreeNode(2);  
const q = new TreeNode(8);  
  
console.log(lowestCommonAncestor(root, p, q).val); // Output: 6
```

## **Explanation:**

### **1. BST Property:**

- Nodes in the left subtree have smaller values than the root.
- Nodes in the right subtree have larger values than the root.

### **2. Recursive Steps:**

- If both p and q are smaller than the root, search the left subtree.
- If both p and q are larger than the root, search the right subtree.
- If one value is on each side, the current node is the LCA.

### **3. Base Case:**

- If the root is null, return null.

### **4. Time Complexity:**

- $O(h)$ : Traversal follows the tree height.

### **5. Space Complexity:**

- $O(h)$ : Space for the recursion stack.

## **Lowest Common Ancestor of a BST (Iterative Approach)**

**312. Problem:** Find the lowest common ancestor (LCA) of two nodes p and q in a binary search tree (BST) using an **iterative approach**.

## **Solution:**

```
function lowestCommonAncestorIterative(root, p, q) {  
    while (root) {  
        if (p.val < root.val && q.val < root.val) {  
            root = root.left; // Move to the left subtree  
        } else if (p.val > root.val && q.val > root.val) {  
            root = root.right; // Move to the right subtree  
        } else {  
            return root; // Current node is the LCA  
        }  
    }  
    return null;  
}  
  
// Test  
const root = new TreeNode(  
    6,  
    new TreeNode(  
        2,  
        new TreeNode(0),  
        new TreeNode(4, new TreeNode(3), new TreeNode(5))  
    ),  
    new TreeNode(8, new TreeNode(7), new TreeNode(9))  
);  
const p = new TreeNode(2);
```

```

const q = new TreeNode(8);

console.log(lowestCommonAncestorIterative(root, p, q).val); // Output: 6

```

### **Explanation:**

#### **1. Iterative Traversal:**

- Start at the root node and iterate down the tree.
- Use the BST properties to determine whether to move left or right.

#### **2. Conditions:**

- If both p and q are smaller than the root, move to the left subtree.
- If both p and q are larger than the root, move to the right subtree.
- Otherwise, the current node is the LCA.

#### **3. Edge Cases:**

- If p or q is the root, return the root as the LCA.
- If the tree is empty (root is null), return null.

#### **4. Time Complexity:**

- $O(h)$ : Traverses the height of the tree.

#### **5. Space Complexity:**

- $O(1)$ : No recursion or additional data structures used.

### **Lowest Common Ancestor of a BST (Binary Search Approach)**

**313. Problem: Use the binary search property of the BST to determine the LCA more efficiently.**

```

function lowestCommonAncestorBinarySearch(root, p, q) {
    while (root) {
        const rootVal = root.val;
        if (p.val < rootVal && q.val < rootVal) {
            root = root.left; // Move left
        } else if (p.val > rootVal && q.val > rootVal) {
            root = root.right; // Move right
        } else {
            return root; // Current node is the LCA
        }
    }
    return null;
}

// Test
const root = new TreeNode(
    6,
    new TreeNode(
        2,
        new TreeNode(0),
        new TreeNode(4, new TreeNode(3), new TreeNode(5))
    ),
)

```

```

new TreeNode(8, new TreeNode(7), new TreeNode(9))
);
const p = new TreeNode(2);
const q = new TreeNode(4);

console.log(lowestCommonAncestorBinarySearch(root, p, q).val); // Output: 2

```

### **Explanation:**

1. **Binary Search Logic:**
  - o The LCA is the first node where p and q split into different subtrees or when one matches the current node.
2. **Conditions:**
  - o Traverse left if both p and q are smaller.
  - o Traverse right if both p and q are larger.
  - o Stop otherwise.
3. **Edge Cases:**
  - o Tree is empty: Return null.
  - o p or q matches the root: Return the root.
4. **Time Complexity:**
  - o  $O(h)$ : Follows the height of the tree.
5. **Space Complexity:**
  - o  $O(1)$ : Iterative traversal requires no extra space.

### **Lowest Common Ancestor of a BST (Optimized with Path Tracking)**

**314. Problem: Track the paths to p and q from the root, then compare the paths to find the LCA.**

### **Solution:**

```

function lowestCommonAncestorWithPaths(root, p, q) {
  function getPath(node, target) {
    const path = [];
    while (node) {
      path.push(node);
      if (target.val < node.val) {
        node = node.left; // Move left
      } else if (target.val > node.val) {
        node = node.right; // Move right
      } else {
        break; // Found the target
      }
    }
    return path;
  }

  const pathP = getPath(root, p);
  const pathQ = getPath(root, q);

```

```

let lca = null;
for (let i = 0; i < Math.min(pathP.length, pathQ.length); i++) {
  if (pathP[i] === pathQ[i]) {
    lca = pathP[i];
  } else {
    break;
  }
}

return lca;
}

// Test
const root = new TreeNode(
  6,
  new TreeNode(
    2,
    new TreeNode(0),
    new TreeNode(4, new TreeNode(3), new TreeNode(5))
  ),
  new TreeNode(8, new TreeNode(7), new TreeNode(9))
);
const p = new TreeNode(2);
const q = new TreeNode(4);

console.log(lowestCommonAncestorWithPaths(root, p, q).val); // Output: 2

```

### Explanation:

1. **Path Tracking:**
  - o Traverse the tree and store the path to p and q.
  - o Paths are stored as arrays of nodes.
2. **Compare Paths:**
  - o Compare the two paths and find the last common node.
3. **Edge Cases:**
  - o If the paths diverge immediately, the root is the LCA.
4. **Time Complexity:**
  - o  $O(h)$ : Traverses the height of the tree to track paths.
5. **Space Complexity:**
  - o  $O(h)$ : Space for the paths.

### Lowest Common Ancestor of a BST (Recursive Optimized with Early Exit)

**315. Problem:** Optimize the recursive approach by adding an early exit condition to return the LCA as soon as it's identified without traversing unnecessary parts of the tree.

```

function lowestCommonAncestorRecursive(root, p, q) {
    if (!root) return null;

    // If both p and q are smaller than root, recurse left
    if (p.val < root.val && q.val < root.val) {
        return lowestCommonAncestorRecursive(root.left, p, q);
    }

    // If both p and q are greater than root, recurse right
    if (p.val > root.val && q.val > root.val) {
        return lowestCommonAncestorRecursive(root.right, p, q);
    }

    // Otherwise, the current node is the LCA
    return root;
}

// Test
const root = new TreeNode(
    6,
    new TreeNode(
        2,
        new TreeNode(0),
        new TreeNode(4, new TreeNode(3), new TreeNode(5))
    ),
    new TreeNode(8, new TreeNode(7), new TreeNode(9))
);
const p = new TreeNode(2);
const q = new TreeNode(4);

console.log(lowestCommonAncestorRecursive(root, p, q).val); // Output: 2

```

### Explanation:

**1. Recursive Traversal:**

- Start at the root and decide whether to move left or right based on the values of p and q.
- Stop as soon as the current node satisfies the condition for being the LCA.

**2. BST Property:**

- Use the property of BSTs where all left descendants are smaller, and all right descendants are larger.

**3. Early Exit:**

- Avoid unnecessary traversal once the LCA is identified.

**4. Time Complexity:**

- $O(h)$ : Traverses the height of the tree.

**5. Space Complexity:**

- $O(h)$ : Space for the recursion stack.

## Lowest Common Ancestor of a BST (Using Morris Traversal)

**316. Problem:** Find the LCA using Morris Traversal to achieve O(1) space.

**Solution:**

```
function lowestCommonAncestorMorris(root, p, q) {  
    let current = root;  
  
    while (current) {  
        if (p.val < current.val && q.val < current.val) {  
            current = current.left; // Move to the left subtree  
        } else if (p.val > current.val && q.val > current.val) {  
            current = current.right; // Move to the right subtree  
        } else {  
            return current; // Current node is the LCA  
        }  
    }  
  
    return null;  
}  
  
// Test  
const root = new TreeNode(  
    6,  
    new TreeNode(  
        2,  
        new TreeNode(0),  
        new TreeNode(4, new TreeNode(3), new TreeNode(5))  
    ),  
    new TreeNode(8, new TreeNode(7), new TreeNode(9))  
);  
const p = new TreeNode(2);  
const q = new TreeNode(8);  
  
console.log(lowestCommonAncestorMorris(root, p, q).val); // Output: 6
```

**Explanation:**

**1. Traversal:**

- Use the BST properties to traverse left or right based on the values of p and q.
- Stop once the current node satisfies the LCA condition.

**2. Space Efficiency:**

- Morris Traversal doesn't require recursion or additional space for a stack.

**3. Time Complexity:**

- $O(h)$ : Traverses the height of the tree.

**4. Space Complexity:**

- $O(1)$ : No extra space is used.

## Lowest Common Ancestor of a BST (Brute Force Using Paths)

**317. Problem:** Use brute force by explicitly finding the paths to p and q and then comparing them to find the LCA.

**Solution:**

```
function lowestCommonAncestorBruteForce(root, p, q) {  
    function findPath(node, target) {  
        const path = [];  
        while (node) {  
            path.push(node);  
            if (target.val < node.val) {  
                node = node.left; // Move left  
            } else if (target.val > node.val) {  
                node = node.right; // Move right  
            } else {  
                break; // Found the target  
            }  
        }  
        return path;  
    }  
  
    const pathP = findPath(root, p);  
    const pathQ = findPath(root, q);  
  
    let lca = null;  
    for (let i = 0; i < Math.min(pathP.length, pathQ.length); i++) {  
        if (pathP[i] === pathQ[i]) {  
            lca = pathP[i];  
        } else {  
            break;  
        }  
    }  
  
    return lca;  
}  
  
// Test  
const root = new TreeNode(  
    6,  
    new TreeNode(  
        2,  
        new TreeNode(0),  
        new TreeNode(4, new TreeNode(3), new TreeNode(5))  
    ),  
    new TreeNode(8, new TreeNode(7), new TreeNode(9))  
);  
const p = new TreeNode(2);
```

```

const q = new TreeNode(4);

console.log(lowestCommonAncestorBruteForce(root, p, q).val); // Output: 2

```

### **Explanation:**

#### **1. Path Traversal:**

- Use a helper function to find the path from the root to a target node.

#### **2. Comparison:**

- Compare the paths to find the last common node, which is the LCA.

#### **3. Time Complexity:**

- $O(h)$ : Finding paths for p and q takes  $O(h)$ , and comparing them also takes  $O(h)$ .

#### **4. Space Complexity:**

- $O(h)$ : Space for storing the paths.

### **Lowest Common Ancestor of a BST (Using Parent Pointers)**

**318. Problem: If the tree nodes have a reference to their parent, find the lowest common ancestor (LCA) of two nodes p and q.**

### **Solution:**

```

function lowestCommonAncestorWithParent(p, q) {
    const ancestors = new Set();

    // Traverse up from p and store its ancestors
    while (p) {
        ancestors.add(p);
        p = p.parent; // Move to the parent
    }

    // Traverse up from q and find the first common ancestor
    while (q) {
        if (ancestors.has(q)) {
            return q; // First common ancestor
        }
        q = q.parent; // Move to the parent
    }

    return null; // No common ancestor found
}

// Test
class TreeNodeWithParent {
    constructor(val, left = null, right = null, parent = null) {
        this.val = val;
        this.left = left;
        this.right = right;
        this.parent = parent;
    }
}

```

```

        this.right = right;
        this.parent = parent;
    }

}

const root = new TreeNodeWithParent(6);
root.left = new TreeNodeWithParent(2, null, null, root);
root.right = new TreeNodeWithParent(8, null, null, root);
root.left.left = new TreeNodeWithParent(0, null, null, root.left);
root.left.right = new TreeNodeWithParent(4, null, null, root.left);

const p = root.left; // Node 2
const q = root.left.right; // Node 4

console.log(lowestCommonAncestorWithParent(p, q).val); // Output: 2

```

### Explanation:

1. **Parent Pointers:**
  - o Each node has a reference to its parent, allowing traversal up the tree.
2. **Ancestor Storage:**
  - o Use a set to store all ancestors of node p.
3. **Finding the LCA:**
  - o Traverse up from node q and return the first common ancestor.
4. **Edge Cases:**
  - o If p or q is null, return null.
5. **Time Complexity:**
  - o  $O(h)$ : Traverse the height of the tree for both p and q.
6. **Space Complexity:**
  - o  $O(h)$ : Space for the set of ancestors.

### Lowest Common Ancestor of a BST (Handling Duplicates)

**319. Problem:** Handle cases where the BST might contain duplicate values. Ensure the solution works for the lowest node with the target value.

### Solution:

```

function lowestCommonAncestorWithDuplicates(root, p, q) {
    if (!root) return null;

    if (p.val < root.val && q.val < root.val) {
        return lowestCommonAncestorWithDuplicates(root.left, p, q); // Move left
    }

    if (p.val > root.val && q.val > root.val) {
        return lowestCommonAncestorWithDuplicates(root.right, p, q); // Move right
    }
}

```

```

        return root; // Current node is the LCA
    }

// Test (Handles duplicates)
const root = new TreeNode(
    6,
    new TreeNode(6, new TreeNode(5), new TreeNode(6)),
    new TreeNode(8, new TreeNode(7), new TreeNode(9))
);
const p = root.left.left; // Node 5
const q = root.left.right; // Node 6 (right child of 6)

console.log(lowestCommonAncestorWithDuplicates(root, p, q).val); // Output: 6

```

### Explanation:

1. **Handling Duplicates:**
  - o Ensure traversal respects the BST property even with duplicate values.
  - o Use the lowest occurrence of a node with the target value as the LCA.
2. **Base Case:**
  - o Return the current node when one node is on the left and the other is on the right, or when the current node matches one of the targets.
3. **Time Complexity:**
  - o  $O(h)$ : Traverse the height of the tree.
4. **Space Complexity:**
  - o  $O(h)$ : Space for the recursion stack.

### Lowest Common Ancestor of a BST (Preorder Traversal)

**320. Problem:** Find the LCA using preorder traversal. Traverse the tree and decide whether to move left, right, or return the current node as the LCA.

### Solution:

```

function lowestCommonAncestorPreorder(root, p, q) {
    if (!root) return null;

    if (root.val === p.val || root.val === q.val) {
        return root; // If the current node matches one of the targets
    }

    const left = lowestCommonAncestorPreorder(root.left, p, q);
    const right = lowestCommonAncestorPreorder(root.right, p, q);

    if (left && right) {
        return root; // If both children return non-null, this is the LCA
    }
}

```

```

        return left || right; // Otherwise, return the non-null child
    }

// Test
const root = new TreeNode(
    6,
    new TreeNode(
        2,
        new TreeNode(0),
        new TreeNode(4, new TreeNode(3), new TreeNode(5))
    ),
    new TreeNode(8, new TreeNode(7), new TreeNode(9))
);
const p = root.left; // Node 2
const q = root.right; // Node 8

console.log(lowestCommonAncestorPreorder(root, p, q).val); // Output: 6

```

### **Explanation:**

#### **1. Preorder Traversal:**

- Visit the root first, then recursively check its left and right subtrees.

#### **2. Returning the LCA:**

- Return the current node if one child returns p and the other returns q.

#### **3. Time Complexity:**

- $O(h)$ : Traverses the height of the tree.

#### **4. Space Complexity:**

- $O(h)$ : Space for the recursion stack.

### **Add and Search Word - Data Structure Design**

**321. Problem: Design a data structure that supports adding new words and searching for a word, including support for wildcard characters (.).**

- **addWord(word): Adds a word to the data structure.**
- **search(word): Searches for a word in the data structure. The word may contain dots (.), where a dot matches any letter.**

### **Solution:**

```

class WordDictionary {
    constructor() {
        this.trie = {} // Root of the Trie
    }

    addWord(word) {
        let node = this.trie;
    }
}

```

```

for (const char of word) {
    if (!node[char]) {
        node[char] = {} // Create a new path for the character
    }
    node = node[char];
}
node.isEnd = true; // Mark the end of a word
}

search(word) {
    const dfs = (node, index) => {
        if (index === word.length) return node.isEnd === true;

        const char = word[index];
        if (char === ".") {
            // Check all possible paths for wildcard
            for (const key in node) {
                if (key !== "isEnd" && dfs(node[key], index + 1)) {
                    return true;
                }
            }
            return false;
        } else {
            // Check specific character path
            if (!node[char]) return false;
            return dfs(node[char], index + 1);
        }
    };
}

return dfs(this.trie, 0);
}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");

console.log(wordDictionary.search("pad")); // Output: false
console.log(wordDictionary.search("bad")); // Output: true
console.log(wordDictionary.search(".ad")); // Output: true
console.log(wordDictionary.search("b..")); // Output: true

```

### Explanation:

#### 1. Trie Structure:

- o Each node in the Trie represents a character.

- The `isEnd` property at a node marks the end of a word.
2. **Add Word:**
    - Traverse the Trie, creating nodes for each character in the word.
    - Mark the end of the word with `isEnd = true`.
  3. **Search Word:**
    - Use DFS to search through the Trie.
    - If the current character is `.`, explore all possible child nodes recursively.
    - If the character is a letter, follow the corresponding path in the Trie.
  4. **Edge Cases:**
    - Search for a word not present in the Trie.
    - Search with multiple wildcards `(.)`.
  5. **Time Complexity:**
    - `addWord`:  $O(n)$ , where  $n$  is the length of the word.
    - `search`:  $O(n * m)$ , where  $n$  is the length of the word and  $m$  is the branching factor in the Trie (average number of child nodes).
  6. **Space Complexity:**
    - $O(n * m)$ : Space for storing the Trie, where  $n$  is the number of words and  $m$  is the average length of words.

### Add and Search Word - Optimized Trie with Class-Based Nodes

**322. Problem: Optimize the `addWord` and `search` methods by using a class to represent each node in the Trie. This improves readability and manages Trie properties more explicitly.**

**Solution:**

```
class TrieNode {
    constructor() {
        this.children = {};// Stores child nodes
        this.isEnd = false;// Marks the end of a word
    }
}

class WordDictionary {
    constructor() {
        this.root = new TrieNode();// Root node of the Trie
    }
}

addWord(word) {
    let node = this.root;
    for (const char of word) {
        if (!node.children[char]) {
            node.children[char] = new TrieNode();// Create new node for the character
        }
        node = node.children[char];
    }
    node.isEnd = true;// Mark the end of the word
}
```

```

search(word) {
  const dfs = (node, index) => {
    if (index === word.length) return node.isEnd; // If the end of the word is reached

    const char = word[index];
    if (char === ".") {
      // Wildcard case: Check all possible children
      for (const key in node.children) {
        if (dfs(node.children[key], index + 1)) {
          return true;
        }
      }
      return false;
    } else {
      // Specific character case: Follow the path
      if (!node.children[char]) return false;
      return dfs(node.children[char], index + 1);
    }
  };
}

return dfs(this.root, 0);
}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("cat");
wordDictionary.addWord("car");
wordDictionary.addWord("cart");

console.log(wordDictionary.search("cat")); // Output: true
console.log(wordDictionary.search("ca.")); // Output: true
console.log(wordDictionary.search("car.")); // Output: false
console.log(wordDictionary.search("c.t")); // Output: true

```

### Explanation:

#### 1. TrieNode Class:

- children: An object storing child nodes for each character.
- isEnd: A boolean flag to mark the end of a word.

#### 2. Add Word:

- Traverse or create nodes for each character.
- Mark the last node as the end of the word.

#### 3. Search Word:

- If the character is ., recursively explore all child nodes.
- If the character is a specific letter, follow its path in the Trie.

#### 4. Edge Cases:

- Empty Trie: Searching for any word returns false.

- Wildcards at different positions in the word (e.g., ..., a..b).

#### 5. Time Complexity:

- addWord:  $O(n)$ , where  $n$  is the length of the word.
- search:  $O(n * m)$ , where  $n$  is the length of the word and  $m$  is the branching factor in the Trie.

#### 6. Space Complexity:

- $O(n * m)$ : Space for the Trie nodes, where  $n$  is the number of words and  $m$  is the average length of the words.

### Add and Search Word - Optimized Wildcard Search

**323. Problem: Further optimize the search process by minimizing the DFS calls when encountering a . wildcard.**

**Solution:**

```
class WordDictionary {
    constructor() {
        this.trie = {};
    }

    addWord(word) {
        let node = this.trie;
        for (const char of word) {
            if (!node[char]) {
                node[char] = {};
            }
            node = node[char];
        }
        node.isEnd = true;
    }

    search(word) {
        const dfs = (node, index) => {
            if (index === word.length) return node.isEnd === true;

            const char = word[index];
            if (char === ".") {
                // Only explore paths with child nodes
                return Object.keys(node).some(
                    (key) => key !== "isEnd" && dfs(node[key], index + 1)
                );
            } else {
                if (!node[char]) return false; // Stop early if path doesn't exist
                return dfs(node[char], index + 1);
            }
        };

        return dfs(this.trie, 0);
    }
}
```

```

    }

}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("dog");
wordDictionary.addWord("dot");
wordDictionary.addWord("dorm");

console.log(wordDictionary.search("d.g")); // Output: true
console.log(wordDictionary.search("do.")); // Output: true
console.log(wordDictionary.search("d.z")); // Output: false
console.log(wordDictionary.search("...")); // Output: true

```

### **Explanation:**

1. **Early Exit:**
  - o For ., only check keys in node and skip further exploration if no valid paths exist.
2. **Efficiency:**
  - o Reduces the number of recursive calls for wildcards, especially in densely populated Trie nodes.
3. **Time Complexity:**
  - o addWord: O(n).
  - o search: O(n \* m), but reduced in practice due to minimized DFS calls.
4. **Space Complexity:**
  - o O(n \* m): Space for the Trie structure.

### **Add and Search Word - Trie with Prefix-Based Search**

**324. Problem:** Extend the functionality of the data structure to support prefix-based search in addition to word-based search.

- **addWord(word): Adds a word to the data structure.**
- **search(word): Searches for a word, including support for . wildcard.**
- **startsWith(prefix): Returns true if there is any word in the data structure that starts with the given prefix.**

### **Solution:**

```

class TrieNode {
  constructor() {
    this.children = {} // Stores child nodes
    this.isEnd = false // Marks the end of a word
  }
}

class WordDictionary {
  constructor() {

```

```

this.root = new TrieNode();
}

addWord(word) {
    let node = this.root;
    for (const char of word) {
        if (!node.children[char]) {
            node.children[char] = new TrieNode(); // Create new node for the character
        }
        node = node.children[char];
    }
    node.isEnd = true; // Mark the end of the word
}

search(word) {
    const dfs = (node, index) => {
        if (index === word.length) return node.isEnd;

        const char = word[index];
        if (char === ".") {
            // Wildcard: Check all possible paths
            for (const key in node.children) {
                if (dfs(node.children[key], index + 1)) {
                    return true;
                }
            }
            return false;
        } else {
            // Specific character: Follow the path
            if (!node.children[char]) return false;
            return dfs(node.children[char], index + 1);
        }
    };
}

return dfs(this.root, 0);
}

startsWith(prefix) {
    let node = this.root;
    for (const char of prefix) {
        if (!node.children[char]) return false; // Prefix not found
        node = node.children[char];
    }
    return true; // Prefix exists
}
}

// Test

```

```

const wordDictionary = new WordDictionary();
wordDictionary.addWord("apple");
wordDictionary.addWord("app");
wordDictionary.addWord("banana");

console.log(wordDictionary.startsWith("app")); // Output: true
console.log(wordDictionary.startsWith("ban")); // Output: true
console.log(wordDictionary.startsWith("bat")); // Output: false
console.log(wordDictionary.search("appl.")); // Output: true
console.log(wordDictionary.search("b.nana")); // Output: true
console.log(wordDictionary.search("....")); // Output: false

```

### Explanation:

1. **TrieNode Class:**
  - o children: Stores child nodes for each character.
  - o isEnd: Marks the end of a word.
2. **Add Word:**
  - o Traverse or create nodes for each character in the word.
  - o Mark the end of the word.
3. **Search Word:**
  - o If the character is ., recursively check all possible paths.
  - o Otherwise, follow the specific character path.
4. **Starts With Prefix:**
  - o Traverse the Trie to ensure the prefix exists.
  - o Return true if the traversal completes successfully.
5. **Edge Cases:**
  - o Empty prefix: Always return true.
  - o Wildcards in search combined with valid and invalid prefixes.
6. **Time Complexity:**
  - o addWord: O(n), where n is the length of the word.
  - o search: O(n \* m), where n is the length of the word and m is the branching factor in the Trie.
  - o startsWith: O(n), where n is the length of the prefix.
7. **Space Complexity:**
  - o O(n \* m): Space for the Trie nodes.

### Add and Search Word - Optimized Memory Usage with Compressed Trie

**325. Problem: Optimize the Trie structure by compressing nodes where possible (e.g., merging single-child paths).**

### Solution:

```

class CompressedTrieNode {
    constructor() {
        this.children = {};  
        // Stores child nodes
        this.isEnd = false;  
        // Marks the end of a word
        this.value = "";  
        // Stores compressed string
    }
}

```

```

        }

}

class CompressedWordDictionary {
    constructor() {
        this.root = new CompressedTrieNode();
    }

    addWord(word) {
        let node = this.root;
        let i = 0;

        while (i < word.length) {
            const char = word[i];
            if (!node.children[char]) {
                const newNode = new CompressedTrieNode();
                newNode.value = word.slice(i); // Store remaining part of the word
                newNode.isEnd = true;
                node.children[char] = newNode;
                return;
            }

            node = node.children[char];

            let commonLength = 0;
            while (
                commonLength < node.value.length &&
                i + commonLength < word.length &&
                word[i + commonLength] === node.value[commonLength]
            ) {
                commonLength++;
            }

            if (commonLength < node.value.length) {
                // Split the node
                const newChild = new CompressedTrieNode();
                newChild.value = node.value.slice(commonLength);
                newChild.children = node.children;
                newChild.isEnd = node.isEnd;

                node.value = node.value.slice(0, commonLength);
                node.children = { [newChild.value[0]]: newChild };
                node.isEnd = commonLength === word.length - i;
            }
        }

        i += commonLength;
    }
}

```

```

node.isEnd = true;
}

search(word) {
  const dfs = (node, index) => {
    if (index === word.length) return node.isEnd;

    const char = word[index];
    if (char === ".") {
      for (const key in node.children) {
        if (dfs(node.children[key], index)) return true;
      }
      return false;
    }

    if (!node.children[char]) return false;
    const child = node.children[char];
    return (
      word.slice(index).startsWith(child.value) &&
      dfs(child, index + child.value.length)
    );
  };

  return dfs(this.root, 0);
}

// Test
const dictionary = new CompressedWordDictionary();
dictionary.addWord("test");
dictionary.addWord("tester");
dictionary.addWord("testing");

console.log(dictionary.search("test")); // Output: true
console.log(dictionary.search("teste.")); // Output: true
console.log(dictionary.search("....")); // Output: false

```

### **Explanation:**

1. **Compressed Nodes:**
  - o Merge single-character paths into a single edge, reducing memory usage.
2. **Time Complexity:**
  - o Similar to traditional Trie but reduced due to compression.
3. **Space Complexity:**
  - o O(n): Reduced space due to compressed paths.

### **Add and Search Word - Using Backtracking for Optimized Search**

**326. Problem: Optimize the search process for handling wildcard characters (.) by using backtracking instead of a pure DFS approach.**

**Solution:**

```
class WordDictionary {
    constructor() {
        this.trie = {} // Root of the Trie
    }

    addWord(word) {
        let node = this.trie;
        for (const char of word) {
            if (!node[char]) {
                node[char] = {} // Create a new path for the character
            }
            node = node[char];
        }
        node.isEnd = true; // Mark the end of the word
    }

    search(word) {
        const backtrack = (node, index) => {
            if (index === word.length) return node.isEnd === true;

            const char = word[index];
            if (char === ".") {
                // Wildcard: Check all possible paths
                for (const key in node) {
                    if (key !== "isEnd" && backtrack(node[key], index + 1)) {
                        return true;
                    }
                }
                return false;
            } else {
                // Specific character
                if (!node[char]) return false;
                return backtrack(node[char], index + 1);
            }
        };
        return backtrack(this.trie, 0);
    }
}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("hello");
wordDictionary.addWord("world");
```

```

wordDictionary.addWord("help");

console.log(wordDictionary.search("h.llo")); // Output: true
console.log(wordDictionary.search("w.rld")); // Output: true
console.log(wordDictionary.search("he.p")); // Output: true
console.log(wordDictionary.search("..llo")); // Output: true
console.log(wordDictionary.search(".....")); // Output: false

```

### **Explanation:**

#### **1. Backtracking Approach:**

- If a wildcard (.) is encountered, iterate through all possible child nodes and recursively check for matches.
- If no match is found, backtrack and try the next path.

#### **2. Search Logic:**

- If the character is a specific letter, follow its path in the Trie.
- If the character is a wildcard, explore all possible child nodes.

#### **3. Edge Cases:**

- Words with multiple wildcards (e.g., ...llo).
- Searching for words longer than any word in the Trie.

#### **4. Time Complexity:**

- addWord: O(n), where n is the length of the word.
- search: O(n \* m), where n is the length of the word and m is the average branching factor in the Trie.

#### **5. Space Complexity:**

- O(n \* m): Space for storing the Trie structure.

### **Add and Search Word - Handling Edge Cases with Empty Words**

#### **327. Problem:**

Handle edge cases where:

- An empty string ("") is added or searched.
- Words consist entirely of wildcards (e.g., "....").

#### **Solution:**

```

class WordDictionary {
    constructor() {
        this.trie = { isEnd: false }; // Root of the Trie
    }

    addWord(word) {
        let node = this.trie;
        for (const char of word) {
            if (!node[char]) {
                node[char] = {}; // Create a new node for the character
            }
            node = node[char];
        }
    }
}

```

```

        }
        node.isEnd = true; // Mark the end of the word
    }

    search(word) {
        const dfs = (node, index) => {
            if (index === word.length) return node.isEnd === true;

            const char = word[index];
            if (char === ".") {
                // Wildcard case: Check all possible children
                for (const key in node) {
                    if (key !== "isEnd" && dfs(node[key], index + 1)) {
                        return true;
                    }
                }
                return false;
            } else {
                if (!node[char]) return false; // Stop if the path doesn't exist
                return dfs(node[char], index + 1);
            }
        };
    }

    return dfs(this trie, 0);
}
}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("");
wordDictionary.addWord("cat");
wordDictionary.addWord("dog");
wordDictionary.addWord("bat");

console.log(wordDictionary.search("")); // Output: true
console.log(wordDictionary.search(".")); // Output: true
console.log(wordDictionary.search("...")); // Output: false
console.log(wordDictionary.search("c.t")); // Output: true
console.log(wordDictionary.search("b.t")); // Output: true

```

## Explanation:

1. **Empty String Handling:**
  - Add logic to mark the root as `isEnd = true` when an empty string is added.
  - Check the root's `isEnd` property during searches for "".
2. **Wildcard Handling:**
  - For words consisting entirely of . (wildcards), recursively check all possible paths.

**3. Edge Cases:**

- Adding or searching for an empty string.
- Searching for words longer than any word in the Trie.

**4. Time Complexity:**

- addWord:  $O(n)$ .
- search:  $O(n * m)$ , reduced for short words or early exits.

**5. Space Complexity:**

- $O(n * m)$ : Space for the Trie.

### **Add and Search Word - Using Two Tries for Word and Reverse Search**

**328. Problem:**

Optimize search functionality by supporting reverse searches. For example:

- Add word: "hello"
- Search for: "olleh" or "e.l.o"

**Solution:**

```
class WordDictionary {  
    constructor() {  
        this.trie = {};  
        this.reverseTrie = {};  
    }  
  
    addWord(word) {  
        let node = this.trie;  
        for (const char of word) {  
            if (!node[char]) node[char] = {};  
            node = node[char];  
        }  
        node.isEnd = true;  
  
        let reverseNode = this.reverseTrie;  
        for (const char of [...word].reverse()) {  
            if (!reverseNode[char]) reverseNode[char] = {};  
            reverseNode = reverseNode[char];  
        }  
        reverseNode.isEnd = true;  
    }  
  
    search(word) {  
        const dfs = (node, index, reverse = false) => {  
            if (index === word.length) return node.isEnd === true;  
  
            const char = word[index];  
            if (reverse) {  
                if (!node[char]) return false;  
                return dfs(node[char], index + 1, reverse);  
            } else {  
                if (!node[char]) return false;  
                return dfs(node[char], index - 1, reverse);  
            }  
        };  
        return dfs(this.trie, 0, false);  
    }  
}
```

```

if (char === ".") {
    // Wildcard: Check all possible paths
    for (const key in node) {
        if (key !== "isEnd" && dfs(node[key], index + 1, reverse)) {
            return true;
        }
    }
    return false;
} else {
    if (!node[char]) return false;
    return dfs(node[char], index + 1, reverse);
}
};

// Determine which Trie to use (normal or reverse)
if (word[0] === "*") {
    const reversedWord = word.slice(1).split("").reverse().join("");
    return dfs(this.reverseTrie, 0, true);
} else {
    return dfs(this.trie, 0);
}
}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("hello");
wordDictionary.addWord("world");
wordDictionary.addWord("help");

console.log(wordDictionary.search("hello")); // Output: true
console.log(wordDictionary.search("olleh")); // Output: true
console.log(wordDictionary.search("wor.d")); // Output: true
console.log(wordDictionary.search("...ld")); // Output: true
console.log(wordDictionary.search("*olleh")); // Output: true

```

## Explanation:

### 1. Two Tries:

- The first Trie stores words normally.
- The second Trie stores words in reverse order for reverse searches.

### 2. Add Word:

- Add the word to the normal Trie.
- Add the reversed word to the reverse Trie.

### 3. Search:

- If the word starts with \*, use the reverse Trie.
- Use DFS to handle wildcard searches (.)

### 4. Edge Cases:

- Reverse wildcard searches like "\*..lleh".
- Searches with only . or \*.

#### 5. Time Complexity:

- addWord: O(n), where n is the length of the word.
- search: O(n \* m), where n is the word length and m is the branching factor.

#### 6. Space Complexity:

- O(2 \* n \* m): Space for the two Tries.

### Add and Search Word - Wildcard Optimization with Limited Depth

**329. Problem: Optimize wildcard searches by limiting the depth of recursion for characters to avoid unnecessary traversal in deep Tries.**

**Solution:**

```
class WordDictionary {
    constructor() {
        this.trie = {};
    }

    addWord(word) {
        let node = this.trie;
        for (const char of word) {
            if (!node[char]) {
                node[char] = {};
            }
            node = node[char];
        }
        node.isEnd = true;
    }

    search(word) {
        const dfs = (node, index, maxDepth) => {
            if (index === word.length) return node.isEnd === true;
            if (maxDepth === 0) return false;

            const char = word[index];
            if (char === ".") {
                for (const key in node) {
                    if (key !== "isEnd" && dfs(node[key], index + 1, maxDepth - 1)) {
                        return true;
                    }
                }
                return false;
            } else {
                if (!node[char]) return false;
                return dfs(node[char], index + 1, maxDepth - 1);
            }
        };
    };
}
```

```

        return dfs(this.trie, 0, word.length);
    }
}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("test");
wordDictionary.addWord("testing");
wordDictionary.addWord("tester");

console.log(wordDictionary.search("t.st")); // Output: true
console.log(wordDictionary.search("t.s.ing")); // Output: false
console.log(wordDictionary.search("t.ster")); // Output: true
console.log(wordDictionary.search("....")); // Output: true

```

### **Explanation:**

1. **Max Depth for Wildcards:**
  - o Track the remaining depth (maxDepth) during DFS to avoid unnecessary traversal.
2. **Search Logic:**
  - o If maxDepth reaches 0 during traversal, stop further recursion.
3. **Time Complexity:**
  - o Reduced compared to naive DFS for deep Tries with wildcards.
4. **Space Complexity:**
  - o  $O(n * m)$ : Same as regular Trie-based implementations.

### **Add and Search Word - Using Map for Compact Representation**

**330. Problem:** Optimize the memory usage of the Trie structure by replacing the plain object with a Map to store child nodes. This provides better performance and reduces key conflicts.

### **Solution:**

```

class TrieNode {
    constructor() {
        this.children = new Map(); // Map for child nodes
        this.isEnd = false; // Marks the end of a word
    }
}

class WordDictionary {
    constructor() {
        this.root = new TrieNode();
    }
}

```

```

addWord(word) {
    let node = this.root;
    for (const char of word) {
        if (!node.children.has(char)) {
            node.children.set(char, new TrieNode()); // Create a new node
        }
        node = node.children.get(char);
    }
    node.isEnd = true; // Mark the end of the word
}

search(word) {
    const dfs = (node, index) => {
        if (index === word.length) return node.isEnd;

        const char = word[index];
        if (char === ".") {
            // Wildcard case: Check all possible paths
            for (const [key, childNode] of node.children) {
                if (dfs(childNode, index + 1)) {
                    return true;
                }
            }
            return false;
        } else {
            if (!node.children.has(char)) return false;
            return dfs(node.children.get(char), index + 1);
        }
    };

    return dfs(this.root, 0);
}

// Test
const wordDictionary = new WordDictionary();
wordDictionary.addWord("cat");
wordDictionary.addWord("cap");
wordDictionary.addWord("cape");

console.log(wordDictionary.search("cat")); // Output: true
console.log(wordDictionary.search("ca.")); // Output: true
console.log(wordDictionary.search("c.p.")); // Output: false
console.log(wordDictionary.search("c.t")); // Output: true
console.log(wordDictionary.search(".")); // Output: false

```

### Explanation:

1. **Using Map:**
  - o Map allows efficient insertion and lookup of child nodes.
  - o Prevents potential key conflicts in objects, especially for special characters.
2. **Add Word:**
  - o Traverse or create nodes for each character using the Map methods.
3. **Search Word:**
  - o If the character is ., recursively check all possible paths in the Map.
  - o If it's a specific letter, follow its path in the Trie using Map.
4. **Edge Cases:**
  - o Words with multiple wildcards (e.g., ...e).
  - o Searching for words longer than any word in the Trie.
5. **Time Complexity:**
  - o addWord: O(n), where n is the length of the word.
  - o search: O(n \* m), where n is the length of the word and m is the average branching factor.
6. **Space Complexity:**
  - o O(n \* m): Reduced slightly compared to plain objects due to Map optimization.

## Word Search II - Using Trie and Backtracking

### 331. Problem:

Given an  $m \times n$  board of characters and a list of strings words, return all words from the list that can be formed by sequentially adjacent letters on the board.

- Words can be constructed using letters in horizontally or vertically adjacent cells.
- Each letter can only be used once per word.

### Solution:

```
class TrieNode {
  constructor() {
    this.children = {} // Stores child nodes
    this.isEnd = false // Marks the end of a word
  }
}

class WordSearch {
  constructor(words) {
    this.root = new TrieNode();
    this.result = new Set(); // Store results to avoid duplicates
    this.buildTrie(words);
  }

  buildTrie(words) {
    for (const word of words) {
      let node = this.root;
      for (const char of word) {
        if (!node.children[char]) {
          node.children[char] = new TrieNode();
        }
        node = node.children[char];
      }
      node.isEnd = true;
    }
  }

  search(board, row, col, node, word) {
    if (node.isEnd) {
      this.result.add(word);
    }
    const directions = [[0, 1], [1, 0], [-1, 0], [0, -1]];
    for (const [dr, dc] of directions) {
      const r = row + dr;
      const c = col + dc;
      if (r < 0 || r >= board.length || c < 0 || c >= board[0].length) {
        continue;
      }
      const nextChar = board[r][c];
      if (node.children[nextChar] === null) {
        continue;
      }
      const nextNode = node.children[nextChar];
      const nextWord = word + nextChar;
      this.search(board, r, c, nextNode, nextWord);
    }
  }

  findWords(board, words) {
    this.buildTrie(words);
    const rows = board.length;
    const cols = board[0].length;
    for (let r = 0; r < rows; r++) {
      for (let c = 0; c < cols; c++) {
        this.search(board, r, c, this.root, '');
      }
    }
    return Array.from(this.result);
  }
}
```

```

        node.children[char] = new TrieNode();
    }
    node = node.children[char];
}
node.isEnd = true; // Mark the end of the word
node.word = word; // Store the word at the end node
}

findWords(board) {
    const rows = board.length;
    const cols = board[0].length;

    const dfs = (row, col, node) => {
        const char = board[row][col];
        if (!node.children[char]) return; // Stop if no matching child in Trie

        const childNode = node.children[char];

        if (childNode.isEnd) {
            this.result.add(childNode.word); // Add word to result
        }

        // Mark the current cell as visited
        board[row][col] = "#";

        // Explore all 4 possible directions
        const directions = [
            [0, 1], // Right
            [0, -1], // Left
            [1, 0], // Down
            [-1, 0], // Up
        ];

        for (const [dx, dy] of directions) {
            const newRow = row + dx;
            const newCol = col + dy;
            if (
                newRow >= 0 &&
                newRow < rows &&
                newCol >= 0 &&
                newCol < cols &&
                board[newRow][newCol] !== "#"
            ) {
                dfs(newRow, newCol, childNode);
            }
        }
    }
}

```

```

// Restore the cell after backtracking
board[row][col] = char;

// Prune the Trie by deleting empty nodes
if (Object.keys(childNode.children).length === 0) {
    delete node.children[char];
}

// Start backtracking from every cell in the board
for (let row = 0; row < rows; row++) {
    for (let col = 0; col < cols; col++) {
        dfs(row, col, this.root);
    }
}

return Array.from(this.result); // Return the results as an array
}

}

// Test
const board = [
    ["o", "a", "a", "n"],
    ["e", "t", "a", "e"],
    ["i", "h", "k", "r"],
    ["i", "f", "l", "v"],
];
const words = ["oath", "pea", "eat", "rain"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["oath", "eat"]

```

## Explanation:

### 1. Trie Construction:

- Build a Trie from the given words list for efficient prefix-based lookups.

### 2. Backtracking:

- Start from each cell in the board.
- Traverse adjacent cells recursively while following the Trie structure.
- Mark cells as visited during the traversal and restore them afterward.

### 3. Pruning:

- Remove child nodes from the Trie if they are no longer needed, reducing search space.

### 4. Edge Cases:

- Empty board or words list.
- Words with overlapping characters on the board.

### 5. Time Complexity:

- $O(n + m * 4^k)$ , where  $n$  is the total number of characters in all words,  $m$  is the number of cells in the board, and  $k$  is the maximum length of a word.

## 6. Space Complexity:

- $O(n)$ : Space for the Trie and recursion stack.

### Word Search II - Using HashSet to Avoid Duplicate Results

**332. Problem:** Optimize the algorithm to avoid duplicate words in the result by using a **HashSet** instead of a list or array.

**Solution:**

```
class WordSearch {
  constructor(words) {
    this.root = new TrieNode();
    this.result = new Set(); // Use a Set to store unique results
    this.buildTrie(words);
  }

  buildTrie(words) {
    for (const word of words) {
      let node = this.root;
      for (const char of word) {
        if (!node.children[char]) {
          node.children[char] = new TrieNode();
        }
        node = node.children[char];
      }
      node.isEnd = true; // Mark the end of the word
      node.word = word; // Store the word at the end node
    }
  }

  findWords(board) {
    const rows = board.length;
    const cols = board[0].length;

    const dfs = (row, col, node) => {
      const char = board[row][col];
      if (!node.children[char]) return; // Stop if no matching child in Trie

      const childNode = node.children[char];

      if (childNode.isEnd) {
        this.result.add(childNode.word); // Add word to result set
      }

      // Mark the current cell as visited
      board[row][col] = "#";
    };
  }
}
```

```

// Explore all 4 possible directions
const directions = [
  [0, 1], // Right
  [0, -1], // Left
  [1, 0], // Down
  [-1, 0], // Up
];

for (const [dx, dy] of directions) {
  const newRow = row + dx;
  const newCol = col + dy;
  if (
    newRow >= 0 &&
    newRow < rows &&
    newCol >= 0 &&
    newCol < cols &&
    board[newRow][newCol] !== "#"
  ) {
    dfs(newRow, newCol, childNode);
  }
}

// Restore the cell after backtracking
board[row][col] = char;

// Prune the Trie by deleting empty nodes
if (Object.keys(childNode.children).length === 0) {
  delete node.children[char];
}
};

// Start backtracking from every cell in the board
for (let row = 0; row < rows; row++) {
  for (let col = 0; col < cols; col++) {
    dfs(row, col, this.root);
  }
}

return Array.from(this.result); // Return the results as an array
}

}

// Test
const board = [
  ["a", "b", "c"],
  ["a", "e", "d"],
  ["a", "f", "g"],
]

```

```
];
const words = ["abcdefg", "gfedcba", "eaabcdgfa", "aefg", "ab"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["abcdefg", "aefg", "ab"]
```

### Explanation:

1. **Trie Construction:**
  - o Build a Trie from the words list.
  - o Use isEnd and word properties for identifying complete words.
2. **Using HashSet:**
  - o Store results in a Set to avoid duplicate entries automatically.
  - o Convert the Set to an array before returning the final results.
3. **Backtracking:**
  - o Traverse adjacent cells recursively while following the Trie structure.
  - o Mark cells as visited to prevent revisiting during the current path.
4. **Pruning the Trie:**
  - o Remove nodes from the Trie once their words are found to optimize further searches.
5. **Edge Cases:**
  - o Board contains overlapping characters for different words.
  - o Words list contains duplicates or words not present on the board.
6. **Time Complexity:**
  - o  $O(n + m * 4^k)$ , where n is the total number of characters in all words, m is the number of cells in the board, and k is the maximum length of a word.
7. **Space Complexity:**
  - o  $O(n)$ : Space for the Trie and recursion stack.

### Word Search II - Early Termination with Trie Pruning

**333. Problem:** Optimize the algorithm further by **pruning the Trie nodes** once the word is found to prevent unnecessary traversals.

### Solution:

```
class WordSearch {
    constructor(words) {
        this.root = new TrieNode();
        this.result = new Set();
        this.buildTrie(words);
    }

    buildTrie(words) {
        for (const word of words) {
            let node = this.root;
            for (const char of word) {
                if (!node.children[char]) {
                    node.children[char] = new TrieNode();
                }
                node = node.children[char];
            }
            node.isEnd = true;
            node.word = word;
        }
    }

    findWords(board) {
        const rows = board.length;
        const cols = board[0].length;
        const visited = new Array(rows).fill(null).map(() => new Array(cols).fill(false));
        const result = new Set();

        for (let i = 0; i < rows; i++) {
            for (let j = 0; j < cols; j++) {
                if (this.search(i, j, board, visited, this.root)) {
                    result.add(board[i][j]);
                }
            }
        }

        return [...result];
    }

    search(x, y, board, visited, node) {
        if (node.isEnd) {
            return true;
        }

        if (x < 0 || x >= board.length || y < 0 || y >= board[0].length || visited[x][y] || !node.children[board[x][y]]) {
            return false;
        }

        visited[x][y] = true;

        const char = board[x][y];
        const childNode = node.children[char];
        if (!childNode) {
            return false;
        }

        if (this.search(x + 1, y, board, visited, childNode) ||
            this.search(x - 1, y, board, visited, childNode) ||
            this.search(x, y + 1, board, visited, childNode) ||
            this.search(x, y - 1, board, visited, childNode)) {
            return true;
        }

        visited[x][y] = false;
        return false;
    }
}
```

```

        }
        node = node.children[char];
    }
    node.isEnd = true;
    node.word = word;
}
}

findWords(board) {
    const rows = board.length;
    const cols = board[0].length;

    const dfs = (row, col, node) => {
        const char = board[row][col];
        if (!node.children[char]) return;

        const childNode = node.children[char];

        if (childNode.isEnd) {
            this.result.add(childNode.word);
            // Remove the word from the Trie to prune it
            delete childNode.word;
            childNode.isEnd = false;
        }
    }

    board[row][col] = "#"; // Mark as visited

    const directions = [
        [0, 1],
        [0, -1],
        [1, 0],
        [-1, 0],
    ];

    for (const [dx, dy] of directions) {
        const newRow = row + dx;
        const newCol = col + dy;
        if (
            newRow >= 0 &&
            newRow < rows &&
            newCol >= 0 &&
            newCol < cols &&
            board[newRow][newCol] !== "#"
        ) {
            dfs(newRow, newCol, childNode);
        }
    }
}

```

```

board[row][col] = char; // Restore cell

// Prune the Trie node if it's empty
if (Object.keys(childNode.children).length === 0) {
  delete node.children[char];
}

for (let row = 0; row < rows; row++) {
  for (let col = 0; col < cols; col++) {
    dfs(row, col, this.root);
  }
}

return Array.from(this.result);
}

// Test
const board = [
  ["o", "a", "b", "n"],
  ["o", "t", "a", "e"],
  ["a", "h", "k", "r"],
  ["a", "f", "l", "v"],
];
const words = ["oath", "pea", "eat", "rain", "oat", "bat"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["oath", "eat", "oat"]

```

### **Explanation:**

1. **Dynamic Trie Pruning:**
  - Words are removed from the Trie after they are found.
  - Reduces unnecessary Trie lookups for already found words.
2. **Improved Backtracking:**
  - Marks cells as visited and restores them after traversal.
3. **Time Complexity:**
  - Similar to the previous solutions,  $O(n + m * 4^k)$ , but reduced in practice due to pruning.
4. **Space Complexity:**
  - $O(n)$ : Space for the Trie and recursion stack.

### **Word Search II - Optimized with Board Preprocessing**

**334. Problem:** Further optimize the algorithm by preprocessing the board to reduce unnecessary Trie lookups for words with characters that are not present on the board.

**Solution:**

```
class WordSearch {
    constructor(words) {
        this.root = new TrieNode();
        this.result = new Set();
        this.buildTrie(words);
    }

    buildTrie(words) {
        for (const word of words) {
            let node = this.root;
            for (const char of word) {
                if (!node.children[char]) {
                    node.children[char] = new TrieNode();
                }
                node = node.children[char];
            }
            node.isEnd = true;
            node.word = word;
        }
    }

    preprocessBoard(board) {
        const frequency = {};
        for (const row of board) {
            for (const char of row) {
                frequency[char] = (frequency[char] || 0) + 1;
            }
        }
        return frequency;
    }

    canWordExist(word, frequency) {
        const wordFrequency = {};
        for (const char of word) {
            wordFrequency[char] = (wordFrequency[char] || 0) + 1;
            if (wordFrequency[char] > (frequency[char] || 0)) {
                return false;
            }
        }
        return true;
    }

    findWords(board) {
        const rows = board.length;
        const cols = board[0].length;
        const boardFrequency = this.preprocessBoard(board);
```

```

const dfs = (row, col, node) => {
  const char = board[row][col];
  if (!node.children[char]) return;

  const childNode = node.children[char];

  if (childNode.isEnd) {
    this.result.add(childNode.word);
    // Remove word from Trie to avoid duplicate searches
    delete childNode.word;
    childNode.isEnd = false;
  }

  board[row][col] = "#"; // Mark as visited

  const directions = [
    [0, 1],
    [0, -1],
    [1, 0],
    [-1, 0],
  ];

  for (const [dx, dy] of directions) {
    const newRow = row + dx;
    const newCol = col + dy;
    if (
      newRow >= 0 &&
      newRow < rows &&
      newCol >= 0 &&
      newCol < cols &&
      board[newRow][newCol] !== "#"
    ) {
      dfs(newRow, newCol, childNode);
    }
  }

  board[row][col] = char; // Restore the board cell

  // Prune the Trie node if it's empty
  if (Object.keys(childNode.children).length === 0) {
    delete node.children[char];
  }
};

// Pre-check all words using board frequency
for (const word in this.root.children) {
  if (!this.canWordExist(word, boardFrequency)) {
    delete this.root.children[word];
  }
}

```

```

        }
    }

    for (let row = 0; row < rows; row++) {
        for (let col = 0; col < cols; col++) {
            dfs(row, col, this.root);
        }
    }

    return Array.from(this.result);
}
}

// Test
const board = [
    ["a", "b", "c"],
    ["a", "e", "d"],
    ["a", "f", "g"],
];
const words = ["abcdefg", "gfedcba", "eaabcdgfa", "aefg", "ab", "zz"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["abcdefg", "aefg", "ab"]

```

### Explanation:

1. **Preprocessing the Board:**
  - o Calculate the frequency of each character on the board.
  - o Use this frequency to skip words that cannot exist on the board.
2. **Early Pruning:**
  - o Skip adding words to the Trie if their characters exceed the board's frequency.
3. **Optimized Backtracking:**
  - o Perform the backtracking search only for valid words.
4. **Time Complexity:**
  - o  $O(b + w + m * 4^k)$ , where b is the number of board cells, w is the number of characters in all words, and m is the number of valid starting cells.
5. **Space Complexity:**
  - o  $O(n)$ : Space for the Trie and recursion stack.

### Word Search II - Iterative Trie Construction and Search

**335. Problem: Improve Trie construction and word search to use iterative methods where possible, reducing recursion overhead.**

### Solution:

```

class WordSearch {
    constructor(words) {

```

```

this.root = new TrieNode();
this.result = new Set();
this.buildTrieIteratively(words);
}

buildTrieIteratively(words) {
  for (const word of words) {
    let node = this.root;
    for (const char of word) {
      if (!node.children[char]) {
        node.children[char] = new TrieNode();
      }
      node = node.children[char];
    }
    node.isEnd = true;
    node.word = word;
  }
}

findWords(board) {
  const rows = board.length;
  const cols = board[0].length;

  const dfs = (row, col, node) => {
    const char = board[row][col];
    if (!node.children[char]) return;

    const childNode = node.children[char];

    if (childNode.isEnd) {
      this.result.add(childNode.word);
      delete childNode.word; // Remove word to avoid duplicates
      childNode.isEnd = false;
    }

    board[row][col] = "#"; // Mark as visited

    const directions = [
      [0, 1],
      [0, -1],
      [1, 0],
      [-1, 0],
    ];

    for (const [dx, dy] of directions) {
      const newRow = row + dx;
      const newCol = col + dy;
      if (

```

```

newRow >= 0 &&
newRow < rows &&
newCol >= 0 &&
newCol < cols &&
board[newRow][newCol] !== "#"
) {
  dfs(newRow, newCol, childNode);
}
}

board[row][col] = char; // Restore the board cell

if (Object.keys(childNode.children).length === 0) {
  delete node.children[char];
}
};

for (let row = 0; row < rows; row++) {
  for (let col = 0; col < cols; col++) {
    dfs(row, col, this.root);
  }
}

return Array.from(this.result);
}
}

// Test
const board = [
  ["o", "a", "b", "n"],
  ["o", "t", "a", "e"],
  ["a", "h", "k", "r"],
  ["a", "f", "l", "v"],
];
const words = ["oath", "pea", "eat", "rain", "oat", "bat"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["oath", "eat", "oat"]

```

## Explanation:

### 1. Character Count in Trie:

- Track the frequency of each character in the words added to the Trie.
- Attach this count to the root node for global use.

### 2. Board Character Preprocessing:

- Compute the frequency of characters on the board.
- Compare it with the Trie's character count before starting the search.

### 3. Backtracking:

- Perform DFS for valid starting points.
- Mark cells as visited during traversal and restore them afterward.

**4. Pruning:**

- Remove words and nodes from the Trie after they are found.

**5. Time Complexity:**

- $O(b + w + m * 4^k)$ , where b is the number of board cells, w is the total characters in words, and m is the number of valid starting cells.

**6. Space Complexity:**

- $O(n)$ : Space for the Trie and recursion stack.

## Word Search II - Optimized Trie with Lazy Trie Pruning

**336. Problem:** Optimize the Trie search by implementing **lazy pruning**, where Trie nodes are only deleted after completing the DFS for a branch.

**Solution:**

```
class WordSearch {
    constructor(words) {
        this.root = new TrieNode();
        this.result = new Set();
        this.buildTrie(words);
    }

    buildTrie(words) {
        for (const word of words) {
            let node = this.root;
            for (const char of word) {
                if (!node.children[char]) {
                    node.children[char] = new TrieNode();
                }
                node = node.children[char];
            }
            node.isEnd = true;
            node.word = word;
        }
    }

    findWords(board) {
        const rows = board.length;
        const cols = board[0].length;

        const dfs = (row, col, node) => {
            const char = board[row][col];
            if (!node.children[char]) return;

            const childNode = node.children[char];

            if (childNode.isEnd) {

```

```

    this.result.add(childNode.word);
    delete childNode.word;
    childNode.isEnd = false;
}

board[row][col] = "#"; // Mark as visited

const directions = [
  [0, 1],
  [0, -1],
  [1, 0],
  [-1, 0],
];

for (const [dx, dy] of directions) {
  const newRow = row + dx;
  const newCol = col + dy;
  if (
    newRow >= 0 &&
    newRow < rows &&
    newCol >= 0 &&
    newCol < cols &&
    board[newRow][newCol] !== "#"
  ) {
    dfs(newRow, newCol, childNode);
  }
}

board[row][col] = char; // Restore the board cell

// Perform lazy pruning
if (Object.keys(childNode.children).length === 0) {
  delete node.children[char];
}
};

for (let row = 0; row < rows; row++) {
  for (let col = 0; col < cols; col++) {
    dfs(row, col, this.root);
  }
}

return Array.from(this.result);
}

// Test
const board = [

```

```

["o", "a", "n"],
["e", "t", "a"],
["i", "h", "k"],
];
const words = ["oath", "pea", "eat", "rain", "oak", "kite"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["oath", "eat"]

```

### Explanation:

1. **Lazy Pruning:**
  - Nodes in the Trie are only deleted if all their children have been processed and no words remain.
  - This ensures the Trie structure remains intact for words sharing common prefixes.
2. **Backtracking with DFS:**
  - Explore all valid adjacent cells recursively.
  - Mark cells as visited during traversal and restore them after returning from recursion.
3. **Efficiency:**
  - By pruning only after completing a branch, unnecessary DFS calls for invalid paths are avoided.
4. **Time Complexity:**
  - $O(b + w + m * 4^k)$ , where b is the board size, w is the total characters in the words, and m is the number of valid starting cells.
5. **Space Complexity:**
  - $O(n)$ : Space for the Trie and recursion stack.

### Word Search II - Multi-Word DFS Optimization

**337. Problem: Optimize the backtracking process by detecting when multiple words are found during a single DFS traversal.**

```

class WordSearch {
    constructor(words) {
        this.root = new TrieNode();
        this.result = new Set();
        this.buildTrie(words);
    }

    buildTrie(words) {
        for (const word of words) {
            let node = this.root;
            for (const char of word) {
                if (!node.children[char]) {
                    node.children[char] = new TrieNode();
                }
            }
        }
    }
}

```

```

        node = node.children[char];
    }
    node.isEnd = true;
    node.word = word;
}
}

findWords(board) {
    const rows = board.length;
    const cols = board[0].length;

    const dfs = (row, col, node) => {
        const char = board[row][col];
        if (!node.children[char]) return;

        const childNode = node.children[char];

        if (childNode.isEnd) {
            this.result.add(childNode.word);
            delete childNode.word;
            childNode.isEnd = false;
        }

        board[row][col] = "#"; // Mark as visited

        const directions = [
            [0, 1],
            [0, -1],
            [1, 0],
            [-1, 0],
        ];

        for (const [dx, dy] of directions) {
            const newRow = row + dx;
            const newCol = col + dy;
            if (
                newRow >= 0 &&
                newRow < rows &&
                newCol >= 0 &&
                newCol < cols &&
                board[newRow][newCol] !== "#"
            ) {
                dfs(newRow, newCol, childNode);
            }
        }

        board[row][col] = char; // Restore the board cell
    }
}

```

```

    if (Object.keys(childNode.children).length === 0) {
      delete node.children[char];
    }
  };

  for (let row = 0; row < rows; row++) {
    for (let col = 0; col < cols; col++) {
      dfs(row, col, this.root);
    }
  }

  return Array.from(this.result);
}

// Test
const board = [
  ["o", "a", "t"],
  ["e", "t", "a"],
  ["i", "h", "k"],
];
const words = ["oath", "eat", "oat", "tea", "hat"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["oath", "eat", "oat"]

```

### **Explanation:**

#### **1. Multi-Word Detection:**

- Check for multiple words during a single DFS traversal.
- Use the word property of Trie nodes to detect all valid matches.

#### **2. Time Complexity:**

- $O(b + w + m * 4^k)$ .

#### **3. Space Complexity:**

- $O(n)$ : Space for the Trie.

## Word Search II - Optimized with Board Clipping

**338. Problem: Further optimize the algorithm by clipping the board to only the rows and columns containing relevant characters from the words.**

### **Solution:**

```

class WordSearch {
  constructor(words) {
    this.root = new TrieNode();
  }
}

```

```

this.result = new Set();
this.buildTrie(words);
}

buildTrie(words) {
  for (const word of words) {
    let node = this.root;
    for (const char of word) {
      if (!node.children[char]) {
        node.children[char] = new TrieNode();
      }
      node = node.children[char];
    }
    node.isEnd = true;
    node.word = word;
  }
}

clipBoard(board, words) {
  const charSet = new Set(words.join(""));
  let minRow = board.length,
    maxRow = 0,
    minCol = board[0].length,
    maxCol = 0;

  for (let row = 0; row < board.length; row++) {
    for (let col = 0; col < board[0].length; col++) {
      if (charSet.has(board[row][col])) {
        minRow = Math.min(minRow, row);
        maxRow = Math.max(maxRow, row);
        minCol = Math.min(minCol, col);
        maxCol = Math.max(maxCol, col);
      }
    }
  }

  // Clip the board to the bounding box
  const clippedBoard = [];
  for (let row = minRow; row <= maxRow; row++) {
    clippedBoard.push(board[row].slice(minCol, maxCol + 1));
  }
  return { clippedBoard, offset: { row: minRow, col: minCol } };
}

findWords(board) {
  const { clippedBoard, offset } = this.clipBoard(
    board,
    Object.keys(this.root.children)
}

```

```

);
const rows = clippedBoard.length;
const cols = clippedBoard[0].length;

const dfs = (row, col, node) => {
  const char = clippedBoard[row][col];
  if (!node.children[char]) return;

  const childNode = node.children[char];

  if (childNode.isEnd) {
    this.result.add(childNode.word);
    delete childNode.word;
    childNode.isEnd = false;
  }

  clippedBoard[row][col] = "#"; // Mark as visited

  const directions = [
    [0, 1],
    [0, -1],
    [1, 0],
    [-1, 0],
  ];

  for (const [dx, dy] of directions) {
    const newRow = row + dx;
    const newCol = col + dy;
    if (
      newRow >= 0 &&
      newRow < rows &&
      newCol >= 0 &&
      newCol < cols &&
      clippedBoard[newRow][newCol] !== "#"
    ) {
      dfs(newRow, newCol, childNode);
    }
  }
}

clippedBoard[row][col] = char; // Restore cell

// Prune empty nodes
if (Object.keys(childNode.children).length === 0) {
  delete node.children[char];
}
};

for (let row = 0; row < rows; row++) {

```

```

        for (let col = 0; col < cols; col++) {
            dfs(row, col, this.root);
        }
    }

    return Array.from(this.result);
}

// Test
const board = [
    ["o", "a", "b", "n"],
    ["e", "t", "a", "e"],
    ["i", "h", "k", "r"],
    ["i", "f", "l", "v"],
];
const words = ["oath", "pea", "eat", "rain", "oat", "bat"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["oath", "eat", "oat"]

```

### **Explanation:**

1. **Board Clipping:**
  - Precompute a bounding box around cells containing relevant characters.
  - Clip the board to this bounding box to reduce the size of the search space.
2. **Backtracking:**
  - Use the clipped board for DFS.
  - Restore cells after backtracking and prune Trie nodes lazily.
3. **Edge Cases:**
  - If none of the words' characters are on the board, return an empty result.
4. **Time Complexity:**
  - Preprocessing:  $O(b)$ , where  $b$  is the total number of cells on the board.
  - Search:  $O(c + m * 4^k)$ , where  $c$  is the size of the clipped board,  $m$  is the number of valid starting points, and  $k$  is the maximum word length.
5. **Space Complexity:**
  - $O(c + n)$ : Space for the clipped board and Trie.

### **Word Search II - Prefix-Based Search Optimization**

**339. Problem: Optimize by terminating the DFS early for prefixes that cannot lead to valid words.**

### **Solution:**

```

class WordSearch {
    constructor(words) {

```

```

this.root = new TrieNode();
this.result = new Set();
this.buildTrie(words);
}

buildTrie(words) {
  for (const word of words) {
    let node = this.root;
    for (const char of word) {
      if (!node.children[char]) {
        node.children[char] = new TrieNode();
      }
      node = node.children[char];
    }
    node.isEnd = true;
    node.word = word;
  }
}

findWords(board) {
  const rows = board.length;
  const cols = board[0].length;

  const dfs = (row, col, node) => {
    const char = board[row][col];
    if (!node.children[char]) return;

    const childNode = node.children[char];

    if (childNode.isEnd) {
      this.result.add(childNode.word);
      delete childNode.word;
      childNode.isEnd = false;
    }

    board[row][col] = "#"; // Mark as visited

    const directions = [
      [0, 1],
      [0, -1],
      [1, 0],
      [-1, 0],
    ];

    for (const [dx, dy] of directions) {
      const newRow = row + dx;
      const newCol = col + dy;
      if (

```

```

newRow >= 0 &&
newRow < rows &&
newCol >= 0 &&
newCol < cols &&
board[newRow][newCol] !== "#"
) {
  dfs(newRow, newCol, childNode);
}
}

board[row][col] = char; // Restore cell

// Prune empty nodes
if (Object.keys(childNode.children).length === 0) {
  delete node.children[char];
}
};

for (let row = 0; row < rows; row++) {
  for (let col = 0; col < cols; col++) {
    dfs(row, col, this.root);
  }
}

return Array.from(this.result);
}
}

// Test
const board = [
  ["a", "b", "c"],
  ["d", "e", "f"],
  ["g", "h", "i"],
];
const words = ["abc", "adg", "aei", "cfi"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findWords(board)); // Output: ["abc", "adg", "aei", "cfi"]

```

## Explanation:

### 1. Dynamic Trie Node Removal:

- After finding a word, delete it from the Trie.
- This prevents revisiting already found words.

### 2. Backtracking:

- Explore all possible paths using DFS while marking cells as visited.
- Restore cells after backtracking to maintain the board state.

### 3. Efficient Search Space Reduction:

- Removing nodes dynamically reduces the Trie size during traversal, minimizing unnecessary lookups.

#### 4. Time Complexity:

- $O(b + w + m * 4^k)$ , where b is the board size, w is the total characters in the words, m is the number of valid starting points, and k is the maximum word length.

#### 5. Space Complexity:

- $O(n)$ : Space for the Trie and recursion stack.

### Word Search II - Optimized with Early Exit for Single Word Completion

**340. Problem: Optimize the Trie-based solution by introducing an early exit for boards where only a single word is required. This reduces unnecessary computations after the first word is found.**

#### Solution:

```
class WordSearch {
    constructor(words) {
        this.root = new TrieNode();
        this.result = new Set();
        this.buildTrie(words);
    }

    buildTrie(words) {
        for (const word of words) {
            let node = this.root;
            for (const char of word) {
                if (!node.children[char]) {
                    node.children[char] = new TrieNode();
                }
                node = node.children[char];
            }
            node.isEnd = true;
            node.word = word;
        }
    }

    findFirstWord(board) {
        const rows = board.length;
        const cols = board[0].length;

        const dfs = (row, col, node) => {
            if (this.result.size > 0) return; // Early exit if a word is found

            const char = board[row][col];
            if (!node.children[char]) return;

            const childNode = node.children[char];
```

```

if (childNode.isEnd) {
    this.result.add(childNode.word);
    return; // Stop further exploration after finding the word
}

board[row][col] = "#"; // Mark as visited

const directions = [
    [0, 1],
    [0, -1],
    [1, 0],
    [-1, 0],
];
;

for (const [dx, dy] of directions) {
    const newRow = row + dx;
    const newCol = col + dy;
    if (
        newRow >= 0 &&
        newRow < rows &&
        newCol >= 0 &&
        newCol < cols &&
        board[newRow][newCol] !== "#"
    ) {
        dfs(newRow, newCol, childNode);
    }
}
;

board[row][col] = char; // Restore the board cell
};

for (let row = 0; row < rows; row++) {
    for (let col = 0; col < cols; col++) {
        dfs(row, col, this.root);
        if (this.result.size > 0) break; // Exit early if a word is found
    }
}

return Array.from(this.result)[0] || null; // Return the first word found or null
}
}

// Test
const board = [
    ["o", "a", "b", "n"],
    ["e", "t", "a", "e"],
    ["i", "h", "k", "r"],
]

```

```

["i", "f", "l", "v"],
];
const words = ["oath", "pea", "eat", "rain", "oat"];

const wordSearch = new WordSearch(words);
console.log(wordSearch.findFirstWord(board)); // Output: "oath"

```

### **Explanation:**

#### **1. Early Exit Mechanism:**

- As soon as a word is found, stop further DFS traversal.
- This reduces unnecessary computations, especially for large boards or long word lists.

#### **2. Backtracking with DFS:**

- Explore valid paths on the board, marking cells as visited and restoring them after backtracking.

#### **3. Efficient Search:**

- Exit DFS immediately when a valid word is found, avoiding traversal of other paths.

#### **4. Time Complexity:**

- $O(b + w + m * 4^k)$ , but reduced in practice due to the early exit, where:
  - b is the size of the board.
  - w is the number of characters in all words.
  - m is the number of valid starting points.
  - k is the maximum length of a word.

#### **5. Space Complexity:**

- $O(n)$ : Space for the Trie and recursion stack.

### **Invert/Flip Binary Tree**

**341. Problem: Given the root of a binary tree, invert the tree by swapping the left and right children of each node and return its root.**

### **Solution:**

#### **Approach 1: Recursive Depth-First Search (DFS)**

```

var invertTree = function (root) {
  if (!root) return null;

  // Swap left and right children
  let temp = root.left;
  root.left = root.right;
  root.right = temp;

  // Recursively invert the left and right subtrees
  invertTree(root.left);
}

```

```
    invertTree(root.right);

    return root;
};
```

### Explanation:

### Objective:

To invert a binary tree, meaning we swap the left and right children of each node in the tree.

### Approach:

We use **recursion** to traverse the tree and swap the left and right child nodes at each level.

### Steps:

1. **Base Case:**
  - o If the root is null, return null because there's nothing to invert.
2. **Swap the Left and Right Subtrees:**
  - o Store the left child in a temporary variable.
  - o Assign the right child to left.
  - o Assign the temporary variable to right.
3. **Recursive Calls:**
  - o Recursively call invertTree on both the left and right subtrees.
  - o This ensures all nodes swap their left and right children.
4. **Return the Root Node:**
  - o After all swaps are done, return the modified root node.

### Why it Works:

Each node swaps its left and right children recursively until the entire tree is processed.

### Time Complexity:

- **O(n)O(n)O(n) (Linear time)** — Each node is visited once.

### Space Complexity:

- **O(h)O(h)O(h) (Recursive stack space)** — The function stack goes as deep as the tree height  $h$ .
- For a **balanced tree**,  $h = O(\log n)$ .
- For a **skewed tree**,  $h = O(n)$ .

## Invert/Flip Binary Tree (Iterative Approach)

**342. Problem:** Given the root of a binary tree, invert the tree by swapping the left and right children of each node and return its root.

### Solution:

#### Approach 2: Iterative Breadth-First Search (BFS) Using a Queue

```
var invertTree = function (root) {
    if (!root) return null;

    let queue = [root];

    while (queue.length > 0) {
        let node = queue.shift();

        // Swap left and right children
        let temp = node.left;
        node.left = node.right;
        node.right = temp;

        // Add children to the queue for processing
        if (node.left) queue.push(node.left);
        if (node.right) queue.push(node.right);
    }

    return root;
};
```

### Explanation:

#### Objective:

To invert a binary tree by swapping left and right children iteratively using a queue.

#### Approach:

We use **Breadth-First Search (BFS)** with a queue to process nodes level by level.

#### Steps:

1. **Base Case:**
  - o If root is null, return null as there is nothing to invert.
2. **Initialize a Queue:**
  - o Start with a queue containing the root node.
3. **Process Each Node:**
  - o Dequeue a node from the front of the queue.
  - o Swap its left and right children.
  - o If the left child exists, enqueue it.
  - o If the right child exists, enqueue it.
4. **Repeat Until the Queue is Empty:**
  - o Continue processing nodes level by level until all nodes are swapped.
5. **Return the Root Node:**

- The tree is now inverted, so return the modified root.

### **Why it Works:**

By processing nodes level by level using BFS, we ensure that every node's children are swapped.

### **Time Complexity:**

- **$O(n)O(n)O(n)$  (Linear time)** — Each node is processed once.

### **Space Complexity:**

- **$O(n)O(n)O(n)$  (Worst case, for a full binary tree, all nodes in a level are stored in the queue).**

### **Invert/Flip Binary Tree (Recursive Approach)**

**343. Problem:** Given the root of a binary tree, invert the tree by swapping the left and right children of each node and return its root.

### **Solution:**

#### **Approach 1: Recursive Depth-First Search (DFS)**

```
var invertTree = function (root) {
    if (!root) return null;

    // Swap left and right subtrees
    let temp = root.left;
    root.left = root.right;
    root.right = temp;

    // Recursively invert left and right subtrees
    invertTree(root.left);
    invertTree(root.right);

    return root;
};
```

### **Explanation:**

### **Objective:**

To invert a binary tree using a **recursive Depth-First Search (DFS)** approach.

### **Approach:**

We recursively swap the left and right children of each node.

## Steps:

1. **Base Case:**
  - o If root is null, return null (nothing to invert).
2. **Swap Children:**
  - o Temporarily store root.left.
  - o Assign root.left = root.right.
  - o Assign root.right = temp.
3. **Recursive Calls:**
  - o Recursively call invertTree on root.left.
  - o Recursively call invertTree on root.right.
4. **Return Root:**
  - o After processing all nodes, return the modified root.

## Why it Works:

- The recursion ensures all nodes are visited and swapped correctly.
- The swapping process inverts the tree efficiently.

## Time Complexity:

- **O(n)** → Each node is visited once.

## Space Complexity:

- **O(h)** → Recursive call stack depth (worst case  $O(n)$  for a skewed tree,  $O(\log n)$  for a balanced tree).

## Invert/Flip Binary Tree (Iterative Approach using BFS)

### 344. Problem Statement:

Given the root of a binary tree, invert the tree by swapping the left and right children of each node and return its root.

### Solution:

#### Approach 2: Iterative Breadth-First Search (BFS)

```
var invertTree = function (root) {  
    if (!root) return null;  
  
    let queue = [root];  
  
    while (queue.length > 0) {  
        let node = queue.shift(); // Remove the front node  
  
        // Swap left and right children
```

```

let temp = node.left;
node.left = node.right;
node.right = temp;

// Add children to the queue for further processing
if (node.left) queue.push(node.left);
if (node.right) queue.push(node.right);
}

return root;
};

```

### **Explanation:**

### **Objective:**

To invert a binary tree using an **iterative Breadth-First Search (BFS) approach**.

### **Approach:**

Use a queue (FIFO) to traverse the tree level by level and swap the left and right children of each node.

### **Steps:**

1. **Edge Case:**
  - o If root is null, return null (nothing to invert).
2. **Initialize Queue:**
  - o Push root into the queue.
3. **Process Nodes in Queue:**
  - o While the queue is **not empty**:
    - Remove the **front** node from the queue.
    - Swap its **left** and **right** children.
    - Push the **new left** and **new right** children into the queue if they exist.
4. **Return Root:**
  - o Once all nodes are processed, return the modified root.

### **Why it Works:**

- The BFS approach ensures every node is visited level by level, making it efficient for large trees.
- The swapping operation is performed in constant time for each node.

### **Time Complexity:**

- **O(n)** → Each node is visited once.

## Space Complexity:

- **O(n)** → In the worst case (when the tree is a full binary tree), the queue can store  $O(n)$  nodes.

## Invert/Flip Binary Tree (Iterative Approach)

### 345. Problem:

Given the root of a binary tree, invert (or flip) the tree using an iterative approach.

The inversion of a binary tree is to swap the left and right child of all nodes in the tree.

Implement the solution using a queue and Breadth-First Search (BFS).

### Example:

Input:

```
4
/\ 
2 7
/\/\
1 3 6 9
```

### Output:

```
4
/\ 
7 2
/\/\
9 6 3 1
```

### Solution:

```
var invertTree = function (root) {
  if (root === null) {
    return null;
  }

  let queue = [root]; // Initialize queue with the root node

  while (queue.length > 0) {
    let currentNode = queue.shift(); // Dequeue the front node

    // Swap the left and right children of the current node
    let temp = currentNode.left;
    currentNode.left = currentNode.right;
    currentNode.right = temp;
```

```

// If left child exists, enqueue it
if (currentNode.left !== null) {
    queue.push(currentNode.left);
}

// If right child exists, enqueue it
if (currentNode.right !== null) {
    queue.push(currentNode.right);
}

return root;
};

```

### **Explanation of Code:**

1. **Queue Initialization:** We start with a queue that contains the root node of the binary tree.
2. **While Loop:**
  - o We use the queue to keep track of the nodes as we process them level by level.
  - o For each node, we:
    - Swap its left and right children.
    - Enqueue the left child if it exists, and enqueue the right child if it exists.
3. **Return the Modified Tree:** After all nodes have been processed, we return the modified tree with its children swapped.

### **Time and Space Complexity:**

- **Time Complexity:**
  - o **O(n):** We traverse each node in the binary tree once, where n is the number of nodes in the tree.
- **Space Complexity:**
  - o **O(n):** The space complexity is proportional to the maximum number of nodes in the queue, which can be O(n) in the worst case.

### **Invert Binary Tree (Using DFS)**

#### **346. Problem:**

Given the root of a binary tree, invert (or flip) the tree using a Depth First Search (DFS) approach.

The inversion of a binary tree is to swap the left and right child of all nodes in the tree.

Implement the solution using **Depth First Search (DFS)**, which involves recursive traversal of the tree.

**Example:**

**Input:**

```
4
/\ 
2 7
/\ / \
1 3 6 9
```

**Output:**

```
4
/\ 
7 2
/\ / \
9 6 3 1
```

**Solution:**

```
var invertTree = function (root) {
    // Base case: if the node is null, return null
    if (root === null) {
        return null;
    }

    // Swap the left and right children of the current node
    let temp = root.left;
    root.left = root.right;
    root.right = temp;

    // Recursively invert the left and right subtrees
    invertTree(root.left);
    invertTree(root.right);

    return root;
};
```

**Explanation of Code:**

1. **Base Case:** If the node is null, return null (the tree or subtree is empty).

2. **Swapping Children:** For the current node, we swap its left and right children using a temporary variable (temp).
3. **Recursive Calls:** After swapping the children, we recursively call invertTree on both the left and right subtrees to invert them.
4. **Return the Root:** Once all nodes have been processed and inverted, we return the modified root of the tree.

### Time Complexity:

- **$O(n)$ :** We visit each node in the binary tree exactly once, where  $n$  is the number of nodes.

### Space Complexity:

- **$O(h)$ :** The space complexity depends on the height  $h$  of the tree, which is the depth of the recursive calls. In the worst case (completely unbalanced tree), the space complexity is  $O(n)$ . For a balanced tree, it is  $O(\log n)$ .

## Invert Binary Tree (Using DFS)

### 347. Problem:

Given the root of a binary tree, invert the tree (flip the tree) and return its root.

### Example:

Input:



Output:



### Solution:

```
var invertTree = function (root) {
```

```

if (root === null) return null; // Base case: if the node is null, return null

// Swap the left and right children
let temp = root.left;
root.left = root.right;
root.right = temp;

// Recursively invert the left and right subtrees
invertTree(root.left);
invertTree(root.right);

return root;
};

```

### **Explanation:**

### **Objective:**

Invert (or flip) the binary tree such that for every node, its left and right children are swapped.

### **Approach:**

#### **1. Recursive DFS Approach:**

- For each node, swap its left and right children.
- Recursively perform the same operation on its left and right children.

#### **2. Steps:**

- Start from the root.
- For each node, if the node is null, return null (base case).
- Swap the left and right children of the node.
- Recursively call the function for the left and right subtrees.

#### **3. Why It Works:**

- The DFS traversal ensures that we visit each node, and by recursively swapping the children, we effectively invert the entire tree.

### **Time Complexity:**

- **O(n):** We traverse each node exactly once, where n is the number of nodes in the tree.

### **Space Complexity:**

- **O(h):** The space complexity is determined by the recursion stack, where h is the height of the tree.

### **Time and Space Complexity:**

- **Time Complexity:**

- **O(n):** We visit each node once to swap the children, where n is the number of nodes in the tree.
- **Space Complexity:**
  - **O(h):** In the worst case (unbalanced tree), the recursion stack depth can be  $O(h)$ , where h is the height of the tree.

## Invert/Flip Binary Tree (Iterative Approach)

### 348. Problem:

**Given the root of a binary tree, invert the tree (flip the tree) and return its root. You need to implement this without recursion, using an iterative approach.**

#### Example:

##### Input:



##### Output:



#### Solution:

```

var invertTree = function (root) {
  if (!root) return null; // If the tree is empty, return null

  let queue = [root]; // Initialize a queue with the root

  while (queue.length > 0) {
    let node = queue.shift(); // Dequeue the current node

    // Swap its left and right children
    let temp = node.left;
    node.left = node.right;
    node.right = temp;
  }
}
  
```

```

    // Enqueue the left and right children if they exist
    if (node.left) queue.push(node.left);
    if (node.right) queue.push(node.right);
}

return root; // Return the root of the inverted tree
};

```

### **Explanation of Code:**

1. **Base Case:**
  - o If the root is null, the function returns null immediately.
2. **Queue Initialization:**
  - o The queue is initialized with the root node of the tree. This will be used to traverse the tree in a level-order fashion.
3. **Main Loop:**
  - o The loop runs until the queue is empty.
  - o For each iteration:
    - The current node is dequeued.
    - The left and right children of the current node are swapped.
    - If the left child exists, it's enqueueued.
    - If the right child exists, it's enqueueued.
4. **Returning the Inverted Tree:**
  - o Once all nodes are processed, the root of the inverted tree is returned.

### **Invert Binary Tree (Using BFS)**

#### **349. Problem:**

Given the root of a binary tree, invert (or flip) the tree using a **Breadth-First Search (BFS)** approach.

The inversion of a binary tree is to swap the left and right child of all nodes in the tree.

Implement the solution using **Breadth-First Search (BFS)**, which uses a queue to traverse the tree level by level.

#### **Example:**

Input:

```

4
/ \
2 7
/\ /\
1 3 6 9

```

### Output:

```
4
/\ 
7 2
/\ \
9 6 3 1
```

### Solution:

```
var invertTree = function (root) {
  if (root === null) {
    return null;
  }

  // Initialize a queue for BFS
  let queue = [root];

  while (queue.length > 0) {
    let node = queue.shift();

    // Swap the left and right children
    let temp = node.left;
    node.left = node.right;
    node.right = temp;

    // Add left and right children to the queue (if not null)
    if (node.left) {
      queue.push(node.left);
    }
    if (node.right) {
      queue.push(node.right);
    }
  }

  return root;
};
```

### Explanation:

### Objective:

Invert the binary tree by swapping the left and right children of every node in the tree using a **Breadth-First Search (BFS)** approach.

## Approach:

1. **Queue-based Traversal (BFS):**
  - o Use a queue to traverse the tree level by level.
  - o For each node in the queue, swap its left and right children.
  - o Add the children of the current node (if not null) to the queue for further processing.
2. **Why It Works:**
  - o BFS ensures that nodes are processed level by level. Swapping the children at each level ensures the tree is fully inverted.

## Time Complexity:

- **O(n):** We visit each node once, where n is the number of nodes in the binary tree.

## Space Complexity:

- **O(n):** In the worst case, the queue will store all nodes at the last level, leading to space complexity of O(n).

## Invert/Flip Binary Tree (Iterative Approach)

### 350. Problem:

Given the root of a binary tree, invert (or flip) the tree using an **Iterative approach**.

The inversion of a binary tree is to swap the left and right child of all nodes in the tree.

Implement the solution iteratively by using a stack to traverse the tree.

### Example:

#### Input:

```
4
/\ 
2 7
/\/\
1 3 6 9
```

#### Output:

```
4
/\ 
7 2
/\/\
9 6 3 1
```

## Solution:

```

var invertTree = function (root) {
  if (root === null) {
    return null;
  }

  let stack = [root];

  while (stack.length > 0) {
    let node = stack.pop();

    // Swap the left and right children
    let temp = node.left;
    node.left = node.right;
    node.right = temp;

    // Push children to the stack (if not null)
    if (node.left) {
      stack.push(node.left);
    }
    if (node.right) {
      stack.push(node.right);
    }
  }

  return root;
};

```

### **Explanation:**

### **Objective:**

Invert the binary tree by swapping the left and right children of every node in the tree using an **Iterative approach**.

### **Approach:**

#### **1. Stack-based Iterative Traversal:**

- Use a stack to perform a depth-first traversal (DFS).
- For each node, swap its left and right children.
- Push the left and right children of the node to the stack for further processing.

#### **2. Why It Works:**

- Using a stack allows for a DFS approach, where each node is processed by swapping its children before moving onto its left and right subtrees.

### **Time Complexity:**

- **O(n):** We visit each node once, where n is the number of nodes in the binary tree.

## Space Complexity:

- **O(n):** In the worst case, the stack will store all nodes, leading to space complexity of  $O(n)$ .

## 4. Dynamic Programming & Optimization Strategies

### Staircase Climbing with Variable Steps (Dynamic Programming)

#### 351. Problem:

You are climbing a staircase. It takes **n** steps to reach the top.

Each time you can either climb **1 step or 2 steps**. In how many distinct ways can you climb to the top?

#### Example 1:

Input: **n = 2**

Output: 2

Explanation: There are two ways to climb to the top:

1. 1 step + 1 step
2. 2 steps

#### Example2:

Input: **n = 3**

Output: 3

Explanation: There are three ways to climb to the top:

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

#### Solution:

```
var climbStairs = function (n) {
    if (n <= 1) return 1; // Base cases for 0 or 1 step

    // Initialize DP array
    let dp = new Array(n + 1);
    dp[0] = 1; // 1 way to stay at ground
    dp[1] = 1; // 1 way to reach the first step

    for (let i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2]; // Ways to reach current step
    }
}
```

```
    return dp[n]; // Total ways to reach the nth step  
};
```

### Explanation:

### Objective:

Find the total number of distinct ways to climb a staircase with n steps when you can take 1 or 2 steps at a time.

### Approach:

#### 1. Dynamic Programming Approach:

- Define  $dp[i]$  as the number of ways to reach the i-th step.
- The relationship is:
  - $dp[i] = dp[i-1] + dp[i-2]$ 
    - From step (i-1), you can take **1 step** to reach i.
    - From step (i-2), you can take **2 steps** to reach i.
- Initialize:
  - $dp[0] = 1$  (1 way to stay at ground)
  - $dp[1] = 1$  (1 way to reach the first step)
- Fill the DP table up to  $dp[n]$ .

#### 2. Why It Works:

- The problem is similar to calculating the Fibonacci sequence, where each number is the sum of the two preceding ones, representing the ways to climb the stairs by taking 1 or 2 steps.

### Time Complexity:

- **O(n):** We calculate the result for each step from 0 to n once.

### Space Complexity:

- **O(n):** We use an array of size n to store the results.

## Staircase Climbing with Variable Steps (Optimized Space Approach)

### 352. Problem:

You are climbing a staircase. It takes **n** steps to reach the top.

Each time you can either climb **1 step** or **2 steps**. In how many distinct ways can you climb to the top?

### Example 1:

Input: **n = 4**

Output: 5

Explanation: There are five ways to climb to the top:

1. 1 step + 1 step + 1 step + 1 step
2. 1 step + 1 step + 2 steps
3. 1 step + 2 steps + 1 step
4. 2 steps + 1 step + 1 step
5. 2 steps + 2 steps

### Solution:

```
function climbStairs(n) {  
    if (n <= 2) return n;  
  
    let first = 1; // Ways to reach the 0th step (base case)  
    let second = 2; // Ways to reach the 1st step (base case)  
  
    for (let i = 3; i <= n; i++) {  
        let current = first + second; // The current step can be reached by summing previous two  
        steps  
        first = second; // Move the first step up  
        second = current; // Move the second step up  
    }  
  
    return second; // The number of ways to reach the nth step  
}  
  
// Example Usage:  
console.log(climbStairs(4)); // Output: 5  
console.log(climbStairs(5)); // Output: 8
```

### Explanation:

### Objective:

Find how many distinct ways there are to reach the top of a staircase with n steps, where you can either climb 1 or 2 steps at a time.

Approach:

#### 1. Base Cases:

- o For n = 1, there is only one way to reach the top (a single 1-step).
- o For n = 2, there are two ways to reach the top (1 step + 1 step or 2 steps).

#### 2. Recurrence Relation:

- o For any n > 2, the number of ways to reach the nth step can be obtained by summing the number of ways to reach the (n-1)th step and the (n-2)th step. This is based on the fact that from the (n-1)th step you can take a single step, and from the (n-2)th step you can take two steps.

### 3. Dynamic Programming Approach:

- We use two variables, first and second, to store the results for the (n-2)th and (n-1)th steps, respectively.
- In each iteration, we compute the number of ways to reach the current step by adding first and second. After that, we update the values of first and second for the next iteration.

### 4. Final Result:

- The result will be stored in the variable second after the loop finishes, which represents the number of ways to reach the nth step.

### Why it Works:

The solution efficiently computes the result by storing intermediate results, thus avoiding redundant computations. This leads to a time complexity of  $O(n)$  and space complexity of  $O(1)$ .

### Time Complexity:

- **$O(n)$**  — The solution iterates from 3 to n once.

### Space Complexity:

- **$O(1)$**  — The solution uses only a constant amount of space to store the previous two results.

## Staircase Climbing with Variable Steps (Recursive Approach with Memoization)

**353. Problem:** You are given a staircase with **n** steps. You can climb **1** step or **2** steps at a time.

Find the number of distinct ways to reach the top using **recursion with memoization**.

---

### Example 1:

Input: **n = 5**

Output: 8

Explanation:

Ways to climb:

1. 1 + 1 + 1 + 1 + 1
2. 1 + 1 + 1 + 2
3. 1 + 1 + 2 + 1
4. 1 + 2 + 1 + 1
5. 2 + 1 + 1 + 1
6. 1 + 2 + 2
7. 2 + 1 + 2
8. 2 + 2 + 1

### Solution:

```
function climbStairs(n, memo = {}) {
```

```

if (n in memo) return memo[n];
if (n === 1) return 1;
if (n === 2) return 2;

memo[n] = climbStairs(n - 1, memo) + climbStairs(n - 2, memo);
return memo[n];
}

console.log(climbStairs(5)); // Output: 8

```

### **Explanation:**

### **Objective:**

Find the number of ways to climb **n** steps using **recursion** while optimizing repeated calculations using **memoization**.

### **Recursive Approach:**

1. The number of ways to reach step **n** is the sum of:
  - o Ways to reach **(n-1)**
  - o Ways to reach **(n-2)**
2. **Recurrence Relation:**

CopyEdit

$$\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$$

3. **Base Cases:**
  - o If  $n == 1$ , return **1**
  - o If  $n == 2$ , return **2**

### **Optimization (Memoization)**

- Store previously computed results in a **hashmap (object)**
- If a value is already computed, return it instead of recalculating.

### **Why It Works:**

- Recursion solves the problem by breaking it into smaller subproblems.
- **Memoization** ensures that we don't compute the same value multiple times, improving efficiency.

### **Time Complexity:**

- **O(n)** (Each unique n is computed only once)

## Space Complexity:

- $O(n)$  (for recursion call stack + memo storage)

### Staircase Climbing with Variable Steps (Iterative Dynamic Programming Approach)

**354. Problem:** You are given a staircase with  $n$  steps. You can take **1 or 2** steps at a time. Find the number of distinct ways to reach the top using an **iterative dynamic programming approach**.

Example 1:

Input:  $n = 6$

Output: 13

Explanation:

The number of ways to reach step 6 is:

1.  $1 + 1 + 1 + 1 + 1 + 1$
2.  $1 + 1 + 1 + 1 + 2$
3.  $1 + 1 + 1 + 2 + 1$
4.  $1 + 1 + 2 + 1 + 1$
5.  $1 + 2 + 1 + 1 + 1$
6.  $2 + 1 + 1 + 1 + 1$
7.  $1 + 1 + 2 + 2$
8.  $1 + 2 + 1 + 2$
9.  $1 + 2 + 2 + 1$
10.  $2 + 1 + 1 + 2$
11.  $2 + 1 + 2 + 1$
12.  $2 + 2 + 1 + 1$
13.  $2 + 2 + 2$

## Solution:

```
function climbStairs(n) {  
    if (n === 1) return 1;  
    if (n === 2) return 2;  
  
    let dp = new Array(n + 1);  
    dp[1] = 1;  
    dp[2] = 2;  
  
    for (let i = 3; i <= n; i++) {  
        dp[i] = dp[i - 1] + dp[i - 2];  
    }  
  
    return dp[n];  
}  
  
console.log(climbStairs(6)); // Output: 13
```

### **Explanation:**

### **Objective:**

Find the number of ways to reach step **n** using **iterative dynamic programming (DP)**.

### **Approach:**

1. **Define an array (dp)** where  $dp[i]$  stores the number of ways to reach step *i*.
2. **Base Cases:**
  - o  $dp[1] = 1$  (One way to climb 1 step: [1])
  - o  $dp[2] = 2$  (Two ways: [1,1] or [2])
3. **Iterate from 3 to n**, filling the dp array using:

$$dp[i] = dp[i - 1] + dp[i - 2];$$

- o This is because from step *i*, you could have arrived from step *i*-1 (1 step) or *i*-2 (2 steps).

4. **Return  $dp[n]$** , which contains the total number of ways to reach the top.

### **Why it Works:**

- The iterative DP approach avoids redundant calculations.
- It builds up the solution step by step, ensuring efficiency.

### **Time Complexity:**

- $O(n)$  → Linear traversal to compute  $dp[n]$ .

### **Space Complexity:**

- $O(n)$  → Extra space for the dp array.

## **Staircase Climbing with Variable Steps (Optimized Space Approach)**

### **355. Problem:**

You are given a staircase with **n** steps. You can take **1** or **2** steps at a time.

Find the number of distinct ways to reach the top using an **optimized space approach**.

### **Example 1:**

Input:  $n = 5$

Output: 8

Explanation:

The number of ways to reach step 5 is:

1. 1 + 1 + 1 + 1 + 1
2. 1 + 1 + 1 + 2
3. 1 + 1 + 2 + 1
4. 1 + 2 + 1 + 1

```
5. 2 + 1 + 1 + 1  
6. 1 + 2 + 2  
7. 2 + 1 + 2  
8. 2 + 2 + 1
```

### Solution:

```
function climbStairs(n) {  
    if (n === 1) return 1;  
    if (n === 2) return 2;  
  
    let prev1 = 2,  
        prev2 = 1,  
        current;  
  
    for (let i = 3; i <= n; i++) {  
        current = prev1 + prev2;  
        prev2 = prev1;  
        prev1 = current;  
    }  
  
    return prev1;  
}  
  
// Example:  
console.log(climbStairs(5)); // Output: 8
```

### Explanation:

#### Objective:

Find the number of ways to climb **n** stairs when you can take **1** or **2** steps at a time, while optimizing space.

#### Approach:

1. Instead of using an array for **dynamic programming (DP)**, we only keep track of the last two computed values.
2. **prev1** stores the number of ways to reach the previous step.
3. **prev2** stores the number of ways to reach the step before that.
4. Iterate from step **3 to n**, updating the values dynamically.

#### Why It Works:

- This is based on the **Fibonacci sequence** where  $dp[i] = dp[i-1] + dp[i-2]$ .
- Instead of storing all values in an array, we only keep track of the last two steps, reducing **space complexity** from  $O(n) \rightarrow O(1)$ .

### Time Complexity:

- **O(n)** → We iterate **n-2** times.

### Space Complexity:

- **O(1)** → Only **two** variables are used instead of an array.

## Staircase Climbing with Variable Steps (Matrix Exponentiation Approach)

### 356. Problem:

You are given a staircase with **n** steps. You can take **1** or **2** steps at a time.

Find the number of distinct ways to reach the top using **Matrix Exponentiation** for an optimized solution.

#### Example 1:

Input: **n = 5**

Output: 8

Explanation:

The number of ways to reach step 5 is:

1. 1 + 1 + 1 + 1 + 1
2. 1 + 1 + 1 + 2
3. 1 + 1 + 2 + 1
4. 1 + 2 + 1 + 1
5. 2 + 1 + 1 + 1
6. 1 + 2 + 2
7. 2 + 1 + 2
8. 2 + 2 + 1

### Solution:

```
function matrixMultiply(A, B) {  
    return [  
        [  
            A[0][0] * B[0][0] + A[0][1] * B[1][0],  
            A[0][0] * B[0][1] + A[0][1] * B[1][1],  
        ],  
        [  
            A[1][0] * B[0][0] + A[1][1] * B[1][0],  
            A[1][0] * B[0][1] + A[1][1] * B[1][1],  
        ],  
    ];  
}  
  
function matrixPower(matrix, n) {  
    let result = [
```

```

[1, 0],
[0, 1],
]; // Identity Matrix
let base = matrix;

while (n > 0) {
  if (n % 2 === 1) {
    result = matrixMultiply(result, base);
  }
  base = matrixMultiply(base, base);
  n = Math.floor(n / 2);
}

return result;
}

function climbStairs(n) {
  if (n === 1) return 1;
  if (n === 2) return 2;

  let transformationMatrix = [
    [1, 1],
    [1, 0],
  ];
  let resultMatrix = matrixPower(transformationMatrix, n - 1);

  return resultMatrix[0][0] + resultMatrix[0][1];
}

// Example:
console.log(climbStairs(5)); // Output: 8

```

### Explanation:

### Objective:

Find the number of ways to climb **n** stairs using **Matrix Exponentiation** to optimize time complexity.

### Approach:

1. The Fibonacci sequence can be represented as matrix exponentiation:  

$$[F(n)F(n-1)] = [1 1 1 0] \times [F(n-1)F(n-2)]$$

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$
2. We use **fast exponentiation** to raise the transformation matrix to the power of **n-1**.
3. This reduces the time complexity to **O(log n)** instead of **O(n)**.

## Why It Works:

- Matrix exponentiation allows us to find the **n-th Fibonacci number** efficiently, which is used to count the number of ways to climb stairs.

**Time Complexity:**  $O(\log n)$  (due to matrix exponentiation).

**Space Complexity:**  $O(1)$  (only a few matrices are stored).

## Staircase Climbing with Variable Steps (Optimized Dynamic Programming)

### 357. Problem:

You are climbing a staircase with **n** steps. Each time you can climb **1 or 2** steps. In how many distinct ways can you reach the top?

Implement an **optimized** dynamic programming approach that reduces space complexity to **O(1)**.

### Solution:

```
function climbStairs(n) {  
    if (n <= 2) return n;  
  
    let prev2 = 1,  
        prev1 = 2;  
  
    for (let i = 3; i <= n; i++) {  
        let current = prev1 + prev2;  
        prev2 = prev1;  
        prev1 = current;  
    }  
  
    return prev1;  
}  
  
// Example Usage:  
console.log(climbStairs(5)); // Output: 8  
console.log(climbStairs(10)); // Output: 89
```

### Explanation:

### Objective:

Find the number of distinct ways to climb **n** stairs using an optimized space **Dynamic Programming** approach.

## Approach:

1. **Understanding the Recurrence Relation:**
  - o The number of ways to reach step **n** is the sum of ways to reach **n-1** and **n-2**:  
$$f(n) = f(n-1) + f(n-2)$$
$$f(n) = f(n-1) + f(n-2)$$
  - o This follows the **Fibonacci sequence**.
2. **Optimized Dynamic Programming (Space Optimization):**
  - o Instead of maintaining an entire DP array (**O(n) space complexity**), we store only the last **two** values (**O(1) space complexity**).
  - o **prev1** represents  $f(n-1)$ , and **prev2** represents  $f(n-2)$ .
  - o At each step, we compute  $\text{current} = \text{prev1} + \text{prev2}$ , then update  $\text{prev2} = \text{prev1}$  and  $\text{prev1} = \text{current}$ .
3. **Time & Space Complexity:**
  - o **Time Complexity:**  $O(n)$  (single loop)
  - o **Space Complexity:**  $O(1)$  (constant space usage)

## Staircase Climbing with Variable Steps (Matrix Exponentiation)

### 358. Problem:

You are given a staircase with **n** steps. Each time you can climb 1 or 2 steps. Find the number of distinct ways to reach the top using Matrix Exponentiation, which optimizes the time complexity.

### Solution:

```
function matrixMultiply(A, B) {  
    return [  
        [  
            A[0][0] * B[0][0] + A[0][1] * B[1][0],  
            A[0][0] * B[0][1] + A[0][1] * B[1][1],  
        ],  
        [  
            A[1][0] * B[0][0] + A[1][1] * B[1][0],  
            A[1][0] * B[0][1] + A[1][1] * B[1][1],  
        ],  
    ];  
}  
  
function matrixPower(matrix, n) {  
    let result = [  
        [1, 0],  
        [0, 1],  
    ]; // Identity matrix  
    while (n > 0) {  
        if (n % 2 === 1) {  
            result = matrixMultiply(result, matrix);  
        }  
        matrix = matrixMultiply(matrix, matrix);  
        n = Math.floor(n / 2);  
    }  
    return result;  
}
```

```

    n = Math.floor(n / 2);
}
return result;
}

function climbStairs(n) {
if (n <= 2) return n;

let baseMatrix = [
[1, 1],
[1, 0],
];
let resultMatrix = matrixPower(baseMatrix, n - 1);

return resultMatrix[0][0] + resultMatrix[0][1];
}

// Example Usage:
console.log(climbStairs(5)); // Output: 8
console.log(climbStairs(10)); // Output: 89

```

### Explanation:

### Objective:

Use **Matrix Exponentiation** to compute the number of ways to climb **n** stairs in **O(log n)** time.

### Approach:

#### 1. Transforming Recurrence into Matrix Form:

- The recurrence relation is:  $f(n) = f(n-1) + f(n-2)$
- This can be represented in matrix form:  

$$[f(n)f(n-1)] = [1 1 0] \times [f(n-1)f(n-2)]$$

$$[f(n)f(n-1)] = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \times [f(n-1)f(n-2)]$$

$$[f(n)f(n-1)] = [1 1 0] \times [f(n-1)f(n-2)]$$

#### 2. Using Matrix Exponentiation:

- Compute the **power of the transformation matrix** efficiently in **O(log n)** time using the **fast exponentiation method**.

#### 3. Final Calculation:

- The top-left value of the resulting matrix gives the final answer.

### Why it Works?

- Reduces **O(n)** time complexity (from iterative DP) to **O(log n)**.
- Efficient for large **n** values.

### Time Complexity:

- **O(log n)** (Matrix Exponentiation)

### Space Complexity:

- **O(1)** (Only constant extra space is used).

## Optimizing Coin Exchange Strategies (Minimum Coins)

### 359. Problem:

You are given an array coins representing coins of different denominations and an integer amount representing a total amount of money. You need to find the **minimum number of coins** that you need to make up that amount. If that amount cannot be made up by any combination of the coins, return **-1**.

### Solution:

```
function coinChange(coins, amount) {  
    let dp = Array(amount + 1).fill(Infinity);  
    dp[0] = 0;  
  
    for (let i = 1; i <= amount; i++) {  
        for (let coin of coins) {  
            if (i - coin >= 0) {  
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);  
            }  
        }  
    }  
  
    return dp[amount] === Infinity ? -1 : dp[amount];  
}  
  
// Example Usage:  
console.log(coinChange([1, 2, 5], 11)); // Output: 3 (5 + 5 + 1)  
console.log(coinChange([2], 3)); // Output: -1
```

### Explanation:

### Objective:

Find the minimum number of coins required to make up the amount using a given list of coin denominations.

### Approach:

1. **Dynamic Programming Approach:**

- We will maintain a DP array  $dp$  where  $dp[i]$  represents the minimum number of coins required to make amount  $i$ .
  - **Initialization:**
    - Set  $dp[0] = 0$  since no coins are needed to make amount 0.
    - Set  $dp[i]$  for all other amounts to Infinity initially, as we don't know the minimum coins yet.
2. **Filling the DP Array:**
- For each coin in coins, iterate over all amounts from coin to amount, and for each amount  $i$ , update the DP value as:  $dp[i] = \min(dp[i], dp[i - \text{coin}] + 1)$
  - The expression  $dp[i - \text{coin}] + 1$  represents the minimum coins needed to make amount  $i - \text{coin}$  plus the current coin.
3. **Final Check:**
- If  $dp[\text{amount}]$  is still Infinity, it means it's not possible to make that amount with the given coins, so return -1. Otherwise, return  $dp[\text{amount}]$  as the answer.

## Why it Works?

- The solution builds up the answer by solving smaller subproblems, and each subproblem involves checking the minimum coins required to make a specific amount, which ensures optimal solutions.

## Time Complexity:

- $O(\text{amount} * \text{coins.length})$  — We iterate through the amount and coins array.

## Space Complexity:

- $O(\text{amount})$  — We maintain an array of size  $\text{amount} + 1$ .

## Staircase Climbing with Variable Steps - Modular Arithmetic for Large Inputs

**360. Problem: For very large inputs n, compute the number of ways to climb n steps modulo  $10^9 + 7$  to avoid overflow.**

## Solution:

```
function climbStairsMod(n) {
  const MOD = 1e9 + 7;
  if (n === 0) return 1;
  if (n === 1) return 1;

  let prev1 = 1; // Ways to climb 1 step
  let prev2 = 1; // Ways to climb 0 steps

  for (let i = 2; i <= n; i++) {
    const current = (prev1 + prev2) % MOD;
    prev2 = prev1;
    prev1 = current;
  }
}
```

```

    return prev1;
}

// Test
console.log(climbStairsMod(10)); // Output: 89
console.log(climbStairsMod(100)); // Output: 573147844013817084101 (modulo 10^9 + 7)
console.log(climbStairsMod(1000)); // Output: 517691607

```

### Explanation:

1. Use two variables `prev1` and `prev2` to store the last two states.
2. Update the values iteratively, applying the modulo operation at each step.
3. **Time Complexity:**
  - o  $O(n)$ : Single loop from 2 to  $n$ .
4. **Space Complexity:**
  - o  $O(1)$ : Constant space for `prev1` and `prev2`.

## Optimizing Coin Exchange Strategies - Minimum Coins to Make Amount

### 361. Problem:

Given an integer array `coins` representing different denominations of coins and an integer `amount`, return the fewest number of coins needed to make up that amount. If it's not possible to make the amount, return `-1`.

```

Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1 (3 coins)

Input: coins = [2], amount = 3
Output: -1

```

### Solution:

```

function coinChange(coins, amount) {
  const dp = new Array(amount + 1).fill(Infinity);
  dp[0] = 0;

  for (const coin of coins) {
    for (let i = coin; i <= amount; i++) {
      dp[i] = Math.min(dp[i], dp[i - coin] + 1);
    }
  }

  return dp[amount] === Infinity ? -1 : dp[amount];
}

```

```
// Test
console.log(coinChange([1, 2, 5], 11)); // Output: 3
console.log(coinChange([2], 3)); // Output: -1
console.log(coinChange([1], 0)); // Output: 0
```

### Explanation:

1. **State Transition:**
  - o For each amount  $i$ , consider all coins. The minimum coins needed for  $i$  is  $dp[i - c] + 1$ , where  $c$  is a coin.
2. **Base Case:**
  - o  $dp[0] = 0$ : No coins needed for amount 0.
3. **Edge Cases:**
  - o If amount is 0, return 0.
  - o If no combination of coins can make the amount, return  $-1$ .

### Time Complexity:

- **$O(n * m)$ :** Where  $n$  is the amount and  $m$  is the number of coins.

### Space Complexity:

- **$O(n)$ :** Space for the  $dp$  array.

## Optimizing Coin Exchange Strategies - Space Optimized

**362. Problem:** Optimize the solution to reduce space usage while maintaining the same logic.

### Solution:

```
function coinChangeOptimized(coins, amount) {
  const dp = new Array(amount + 1).fill(Infinity);
  dp[0] = 0;

  for (const coin of coins) {
    for (let i = coin; i <= amount; i++) {
      dp[i] = Math.min(dp[i], dp[i - coin] + 1);
    }
  }

  return dp[amount] === Infinity ? -1 : dp[amount];
}

// Test
```

```
console.log(coinChangeOptimized([1, 2, 5], 11)); // Output: 3
console.log(coinChangeOptimized([2], 3)); // Output: -1
console.log(coinChangeOptimized([186, 419, 83, 408], 6249)); // Output: 20
```

### Explanation:

Use a single-dimensional array as in the original solution, but modify it in-place to save space.

### Time Complexity:

- $O(n * m)$ : Same as before.

### Space Complexity:

- $O(n)$ : Single array of size `amount + 1`.

## Optimizing Coin Exchange Strategies - Recursive + Memoization

**363. Problem:** Use a top-down approach with memoization to compute the minimum coins recursively.

### Solution:

```
function coinChangeRecursive(coins, amount) {
    const memo = {};

    function helper(remaining) {
        if (remaining === 0) return 0;
        if (remaining < 0) return Infinity;

        if (memo[remaining] !== undefined) return memo[remaining];

        let minCoins = Infinity;
        for (const coin of coins) {
            const result = helper(remaining - coin);
            if (result !== Infinity) {
                minCoins = Math.min(minCoins, result + 1);
            }
        }

        memo[remaining] = minCoins;
        return minCoins;
    }
}
```

```

    const result = helper(amount);
    return result === Infinity ? -1 : result;
}

// Test
console.log(coinChangeRecursive([1, 2, 5], 11)); // Output: 3
console.log(coinChangeRecursive([2], 3)); // Output: -1
console.log(coinChangeRecursive([1], 0)); // Output: 0

```

### Explanation:

#### Recursive with Memoization

1. Define a recursive function `helper(amount)`:
  - o Base case: If `amount === 0`, return 0. If `amount < 0`, return `Infinity`.
  - o Otherwise, compute the minimum coins needed using recursion.
2. Use a `memo` object to store results for each amount.

#### Time Complexity:

- **$O(n * m)$** : Each state is computed once and stored in the memo.

#### Space Complexity:

- **$O(n)$** : Space for memoization and recursion stack.

### Optimizing Coin Exchange Strategies - Maximum Number of Coins for an Amount

**364. Problem:** Find the maximum number of coins that can be used to make a given amount using the denominations in `coins`. If it's not possible to make the amount, return **-1**.

#### Solution:

```

function maxCoinsChange(coins, amount) {
  const dp = new Array(amount + 1).fill(-Infinity);
  dp[0] = 0;

  for (const coin of coins) {
    for (let i = coin; i <= amount; i++) {
      dp[i] = Math.max(dp[i], dp[i - coin] + 1);
    }
  }

  return dp[amount] === -Infinity ? -1 : dp[amount];
}

```

```
// Test
console.log(maxCoinsChange([1, 2, 5], 11)); // Output: 11 (all coins
are 1)
console.log(maxCoinsChange([2], 3)); // Output: -1 (not possible)
console.log(maxCoinsChange([1, 3], 7)); // Output: 7 (all coins are
1)
```

### Explanation:

1. Initialize `dp[i]` to `-Infinity` for amounts that can't be reached.
2. Update the `dp` table iteratively for each coin, keeping track of the maximum number of coins.

### Time Complexity:

- **$O(n * m)$ :** Where  $n$  is the amount and  $m$  is the number of coins.

### Space Complexity:

- **$O(n)$ :** Space for the `dp` array.

## Optimizing Coin Exchange Strategies - Count Total Ways to Make Amount

**365. Problem: Find the total number of distinct ways to make up a given amount using the denominations in coins.**

### Solution:

```
function countWaysToMakeAmount(coins, amount) {
  const dp = new Array(amount + 1).fill(0);
  dp[0] = 1;

  for (const coin of coins) {
    for (let i = coin; i <= amount; i++) {
      dp[i] += dp[i - coin];
    }
  }

  return dp[amount];
}

// Test
console.log(countWaysToMakeAmount([1, 2, 5], 5)); // Output: 4
// Explanation: [5], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]
console.log(countWaysToMakeAmount([2], 3)); // Output: 0
console.log(countWaysToMakeAmount([10], 10)); // Output: 1
```

### Explanation:

1. Each coin contributes to all amounts greater than or equal to its value.
2. Sum up the number of ways to reach the current amount using the previous coins.

### Time Complexity:

- $O(n * m)$ : Each amount is updated for every coin.

### Space Complexity:

- $O(n)$ : Space for the dp array.

## Optimizing Coin Exchange Strategies - Minimum Coins for Exact Amount

**366. Problem: What if you can only use each coin a limited number of times? Find the minimum coins required to make the amount.**

### Solution:

```
function coinChangeLimited(coins, amount, limits) {  
    const dp = new Array(amount + 1).fill(Infinity);  
    dp[0] = 0;  
  
    for (let i = 0; i < coins.length; i++) {  
        const coin = coins[i];  
        const limit = limits[i];  
  
        for (let j = amount; j >= coin; j--) {  
            for (let k = 1; k <= limit && j - k * coin >= 0; k++) {  
                dp[j] = Math.min(dp[j], dp[j - k * coin] + k);  
            }  
        }  
    }  
  
    return dp[amount] === Infinity ? -1 : dp[amount];  
}  
  
// Test  
console.log(coinChangeLimited([1, 2, 5], 11, [10, 2, 1])); // Output: 5  
// Explanation: [1, 1, 1, 2, 5]  
console.log(coinChangeLimited([2, 3], 7, [1, 1])); // Output: -1  
console.log(coinChangeLimited([1, 2, 3], 5, [1, 2, 1])); // Output: 3
```

### Explanation:

1. For each coin, iterate backward to ensure the usage limit is respected.
2. Update the dp table only if the current usage of the coin does not exceed its limit.

### Time Complexity:

- **O(n \* m \* l)**: Where n is the amount, m is the number of coins, and l is the maximum limit for any coin.

### Space Complexity:

- **O(n)**: Space for the dp array.

## Optimizing Coin Exchange Strategies - Finding the Best Combination Selections for Optimizing Coin Exchange Strategies

**367. Problem: Find all combinations of coins that sum up to the given amount. Return all possible unique combinations.**

### Solution:

```
function combinationSum(coins, amount) {
  const result = [];

  function backtrack(start, remaining, combination) {
    if (remaining === 0) {
      result.push([...combination]);
      return;
    }
    if (remaining < 0) return;

    for (let i = start; i < coins.length; i++) {
      combination.push(coins[i]);
      backtrack(i, remaining - coins[i], combination); // Allow same coin
      combination.pop(); // Backtrack
    }
  }

  backtrack(0, amount, []);
  return result;
}

// Test
console.log(combinationSum([1, 2, 5], 5));
// Output: [[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 2, 2], [5]]
console.log(combinationSum([2, 3], 7));
// Output: [[2, 2, 3]]
console.log(combinationSum([2, 4], 6));
// Output: [[2, 2, 2], [2, 4]]
```

## Explanation:

1. Use backtracking to explore all paths where the sum equals the target amount.
2. Ensure unique combinations by starting from the current index `start`.

## Time Complexity:

- $O(2^n)$ : Exponential due to backtracking.

## Space Complexity:

- $O(n)$ : Space for the recursion stack.

## Optimizing Coin Exchange Strategies - Fewest Coins with Infinite Supply

**368. Problem:** Find the fewest coins required to make the amount, assuming there's an infinite supply of each coin. However, return the **exact coin combinations** for the solution.

## Solution:

```
function coinChangeWithCombination(coins, amount) {  
    const dp = new Array(amount + 1).fill(Infinity);  
    const combinations = new Array(amount + 1).fill(null);  
  
    dp[0] = 0;  
    combinations[0] = [];  
  
    for (const coin of coins) {  
        for (let i = coin; i <= amount; i++) {  
            if (dp[i - coin] + 1 < dp[i]) {  
                dp[i] = dp[i - coin] + 1;  
                combinations[i] = [...combinations[i - coin], coin];  
            }  
        }  
    }  
  
    return dp[amount] === Infinity ? -1 : combinations[amount];  
}  
  
// Test  
console.log(coinChangeWithCombination([1, 2, 5], 11)); // Output:  
[5, 5, 1]  
console.log(coinChangeWithCombination([2], 3)); // Output: -1  
console.log(coinChangeWithCombination([1], 7)); // Output: [1, 1, 1,  
1, 1, 1]
```

### Explanation:

1. The `combinations` array stores the exact combination of coins used to make each amount.
2. Update the combination whenever a smaller number of coins is found.

### Time Complexity:

- $O(n * m)$ : Each coin iterates through all amounts.

### Space Complexity:

- $O(n)$ : Space for `dp` and `combinations`.

## Optimizing Coin Exchange Strategies - Minimum Coins Using BFS

**369. Problem:** Find the fewest coins required using a **Breadth-First Search (BFS)** approach.

### Solution:

```
function coinChangeBFS(coins, amount) {  
    if (amount === 0) return 0;  
  
    const queue = [amount];  
    const visited = new Set();  
    let steps = 0;  
  
    while (queue.length > 0) {  
        const size = queue.length;  
        steps++;  
  
        for (let i = 0; i < size; i++) {  
            const current = queue.shift();  
  
            for (const coin of coins) {  
                const next = current - coin;  
  
                if (next === 0) return steps;  
                if (next > 0 && !visited.has(next)) {  
                    queue.push(next);  
                    visited.add(next);  
                }  
            }  
        }  
    }  
  
    return -1;  
}
```

```
// Test
console.log(coinChangeBFS([1, 2, 5], 11)); // Output: 3
console.log(coinChangeBFS([2], 3)); // Output: -1
console.log(coinChangeBFS([186, 419, 83, 408], 6249)); // Output: 20
```

### Explanation:

1. Each level in the BFS represents the number of coins used.
2. Terminate as soon as the target amount is reached.

### Time Complexity:

- **O(n \* m)**: Where n is the amount and m is the number of coins.

### Space Complexity:

- **O(n)**: Space for the queue and visited set.

## Optimizing Coin Exchange Strategies - Count Ways Using Recursion

**370. Problem:** Count the total number of ways to make the given amount using recursion. Return the count of distinct ways.

### Solution:

```
function countWaysRecursive(coins, amount, index = 0) {
    if (amount === 0) return 1; // Found a valid combination
    if (amount < 0 || index === coins.length) return 0; // Invalid case

    // Include the current coin or skip it
    return (
        countWaysRecursive(coins, amount - coins[index], index) +
        countWaysRecursive(coins, amount, index + 1)
    );
}

// Test
console.log(countWaysRecursive([1, 2, 5], 5)); // Output: 4
console.log(countWaysRecursive([2], 3)); // Output: 0
console.log(countWaysRecursive([2, 4], 6)); // Output: 2
```

### Explanation:

1. Base case:
  - If amount == 0, count it as one way.

- o If `amount < 0` or no coins remain, terminate the path.
2. Recursive step:
- o Include the current coin (`amount - coins[index]`).
  - o Skip the current coin (`index + 1`).

### Time Complexity:

- **O(2^n)**: Exponential due to recursive branching.

### Space Complexity:

- **O(n)**: Space for the recursion stack.

## Longest Increasing Subsequence - Using Dynamic Programming

**371. Problem:** Given an array `nums`, return the length of the longest strictly increasing subsequence.

```
Input: nums = [10, 9, 2, 5, 3, 7, 101, 18]
Output: 4
Explanation: The longest increasing subsequence is [2, 3, 7, 101],
and its length is 4.
```

### Solution:

```
function lengthOfLIS(nums) {
  const n = nums.length;
  const dp = new Array(n).fill(1);

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

  return Math.max(...dp);
}

// Test
console.log(lengthOfLIS([10, 9, 2, 5, 3, 7, 101, 18])); // Output: 4
console.log(lengthOfLIS([0, 1, 0, 3, 2, 3])); // Output: 4
console.log(lengthOfLIS([7, 7, 7, 7])); // Output: 1
```

### Explanation:

- Use a **Dynamic Programming (DP)** approach.
- Define  $dp[i]$  as the length of the longest increasing subsequence ending at index  $i$ .
- For each  $i$ , check all previous indices  $j$  ( $j < i$ ) such that  $nums[j] < nums[i]$  and update  $dp[i] = \max(dp[i], dp[j] + 1)$ .
- Return the maximum value in the  $dp$  array.

### Time Complexity:

- **$O(n^2)$** : Two nested loops iterate through the array.

### Space Complexity:

- **$O(n)$** : Space for the  $dp$  array.

## Longest Increasing Subsequence - Using Binary Search

**372. Problem: Optimize the solution to run in  $O(n \log n)$  time.**

### Solution:

```
function lengthOfLIS(nums) {
  const sub = [];

  for (const num of nums) {
    let left = 0,
        right = sub.length;
    while (left < right) {
      const mid = Math.floor((left + right) / 2);
      if (sub[mid] < num) {
        left = mid + 1;
      } else {
        right = mid;
      }
    }
    if (left < sub.length) {
      sub[left] = num;
    } else {
      sub.push(num);
    }
  }

  return sub.length;
}

// Test
console.log(lengthOfLIS([10, 9, 2, 5, 3, 7, 101, 18])); // Output: 4
console.log(lengthOfLIS([0, 1, 0, 3, 2, 3])); // Output: 4
console.log(lengthOfLIS([7, 7, 7, 7])); // Output: 1
```

### Explanation:

1. Use a list (`sub`) to track the smallest ending element of increasing subsequences of various lengths.
2. For each element in `nums`, perform a binary search on `sub`:
  - o If the element is larger than all elements in `sub`, append it.
  - o Otherwise, replace the first element in `sub` that is larger than or equal to it.
3. The length of `sub` at the end is the length of the LIS.

### Time Complexity:

- **$O(n \log n)$** : Each binary search takes  $O(\log n)$ .

### Space Complexity:

- **$O(n)$** : Space for the `sub` array.

## Longest Increasing Subsequence - Print the Subsequence

### 373. Problem: Find and return the actual longest increasing subsequence.

#### Solution:

```
function findLIS(nums) {
  const n = nums.length;
  const dp = new Array(n).fill(1);
  const prev = new Array(n).fill(-1);
  let maxIndex = 0;

  for (let i = 1; i < n; i++) {
    for (let j = 0; j < i; j++) {
      if (nums[j] < nums[i] && dp[j] + 1 > dp[i]) {
        dp[i] = dp[j] + 1;
        prev[i] = j;
      }
    }
    if (dp[i] > dp[maxIndex]) {
      maxIndex = i;
    }
  }

  const lis = [];
  for (let i = maxIndex; i !== -1; i = prev[i]) {
    lis.unshift(nums[i]);
  }

  return lis;
}

// Test
```

```

console.log(findLIS([10, 9, 2, 5, 3, 7, 101, 18])); // Output: [2, 3, 7, 101]
console.log(findLIS([0, 1, 0, 3, 2, 3])); // Output: [0, 1, 2, 3]
console.log(findLIS([7, 7, 7, 7])); // Output: [7]

```

### Explanation:

1. Use the DP approach to calculate the LIS length while maintaining a `prev` array to reconstruct the subsequence.
2. At each `i`, track the index of the previous element in the LIS.

### Time Complexity:

- $O(n^2)$ : Two nested loops for DP computation.

### Space Complexity:

- $O(n)$ : Space for `dp` and `prev` arrays.

## Longest Increasing Subsequence - Recursive with Memoization

### 374. Problem: Solve LIS using a top-down recursive approach with memoization.

#### Solution:

```

function lisRecursive(nums) {
  const memo = new Map();

  function helper(idx, prev) {
    if (idx === nums.length) return 0;

    const key = `${idx}-${prev}`;
    if (memo.has(key)) return memo.get(key);

    let take = 0;
    if (prev < nums[idx]) {
      take = 1 + helper(idx + 1, nums[idx]);
    }
    const skip = helper(idx + 1, prev);

    memo.set(key, Math.max(take, skip));
    return memo.get(key);
  }

  return helper(0, -Infinity);
}

```

```
// Test
console.log(lisRecursive([10, 9, 2, 5, 3, 7, 101, 18])); // Output: 4
console.log(lisRecursive([0, 1, 0, 3, 2, 3])); // Output: 4
console.log(lisRecursive([7, 7, 7, 7])); // Output: 1
```

### Explanation:

1. Define a recursive function `lis(idx, prev)`:
  - o `idx` is the current index.
  - o `prev` is the previous element in the LIS.
2. Memoize results for `(idx, prev)` pairs to avoid redundant calculations.

### Time Complexity:

- $O(n^2)$ : Each state is computed once.

### Space Complexity:

- $O(n^2)$ : Space for memoization.

## Longest Increasing Subsequence - Optimized Recursive with Binary Search

**375. Problem: Optimize the recursive solution using binary search to improve performance while maintaining correctness.**

### Solution:

```
function lisBinaryRecursive(nums) {
  const sub = [];

  function binarySearchInsert(num) {
    let left = 0,
        right = sub.length;
    while (left < right) {
      const mid = Math.floor((left + right) / 2);
      if (sub[mid] < num) {
        left = mid + 1;
      } else {
        right = mid;
      }
    }
    return left;
  }

  function helper(idx) {
    if (idx === nums.length) return sub.length;
    const num = nums[idx];
    const insertIndex = binarySearchInsert(num);
    sub[insertIndex] = num;
    return helper(idx + 1);
  }
  return helper(0);
}
```

```

const pos = binarySearchInsert(nums[idx]);
const temp = sub[pos];
if (pos < sub.length) {
    sub[pos] = nums[idx];
} else {
    sub.push(nums[idx]);
}

const result = helper(idx + 1);
if (temp !== undefined) {
    sub[pos] = temp;
} else {
    sub.pop();
}

return result;
}

return helper(0);
}

// Test
console.log(lisBinaryRecursive([10, 9, 2, 5, 3, 7, 101, 18])); // Output: 4
console.log(lisBinaryRecursive([0, 1, 0, 3, 2, 3])); // Output: 4
console.log(lisBinaryRecursive([7, 7, 7, 7])); // Output: 1

```

### Explanation:

1. Use a list (`sub`) to track the smallest ending elements of increasing subsequences of various lengths.
2. Implement the recursive logic while maintaining the list using binary search for efficient updates.

### Time Complexity:

- $O(n \log n)$ : Binary search for each element.

### Space Complexity:

- $O(n)$ : Space for the `sub` list.

## Longest Increasing Subsequence - Find All Subsequences

**376. Problem:** **Find all increasing subsequences (not just the longest one) in the given array.**

### Solution:

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

  function backtrack(start, path) {
    if (path.length > 1) result.push([...path]);

    for (let i = start; i < nums.length; i++) {
      if (path.length === 0 || nums[i] > path[path.length - 1]) {
        path.push(nums[i]);
        backtrack(i + 1, path);
        path.pop();
      }
    }
  }

  backtrack(0, []);
  return result;
}

// Test
console.log(findAllLIS([4, 6, 7, 7]));
// Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7, 7]]
console.log(findAllLIS([1, 2, 3]));
// Output: [[1, 2], [1, 3], [1, 2, 3], [2, 3]]
```

### Explanation:

1. Use backtracking to explore all subsequences.
2. Include each element in the path only if it maintains the increasing order.

### Time Complexity:

- **O( $2^n$ ):** Exponential due to backtracking.

### Space Complexity:

- **O(n):** Space for the recursion stack.

## Longest Increasing Subsequence - Using Fenwick Tree

**377. Problem: Use a Fenwick Tree (or Binary Indexed Tree) to efficiently calculate the length of the LIS for large input sizes.**

### Solution:

```
function lengthOfLISFenwick(nums) {
    const sorted = [...new Set(nums)].sort((a, b) => a - b);
    const rankMap = new Map(sorted.map((val, idx) => [val, idx + 1]));
    const fenwick = new Array(sorted.length + 1).fill(0);

    function update(index, value) {
        while (index < fenwick.length) {
            fenwick[index] = Math.max(fenwick[index], value);
            index += index & -index;
        }
    }

    function query(index) {
        let max = 0;
        while (index > 0) {
            max = Math.max(max, fenwick[index]);
            index -= index & -index;
        }
        return max;
    }

    let maxLength = 0;
    for (const num of nums) {
        const rank = rankMap.get(num);
        const currentLength = query(rank - 1) + 1;
        update(rank, currentLength);
        maxLength = Math.max(maxLength, currentLength);
    }

    return maxLength;
}

// Test
console.log(lengthOfLISFenwick([10, 9, 2, 5, 3, 7, 101, 18])); // Output: 4
console.log(lengthOfLISFenwick([0, 1, 0, 3, 2, 3])); // Output: 4
console.log(lengthOfLISFenwick([7, 7, 7, 7])); // Output: 1
```

### Explanation:

1. Map array values to ranks for indexing the Fenwick Tree.
2. Use the Fenwick Tree to efficiently compute the LIS length by updating and querying ranks.

### Time Complexity:

- **O(n log n)**: Logarithmic updates and queries for each element.

### Space Complexity:

- **O(n)**: Space for the Fenwick Tree.

### Longest Increasing Subsequence - Using Segment Tree

**378. Problem: Use a Segment Tree to compute the length of the Longest Increasing Subsequence efficiently for larger input sizes.**

#### Solution:

```
class SegmentTree {
    constructor(size) {
        this.tree = new Array(4 * size).fill(0);
    }

    query(start, end, left, right, node) {
        if (left > end || right < start) return 0; // Out of range
        if (left <= start && end <= right) return this.tree[node]; // Fully within range

        const mid = Math.floor((start + end) / 2);
        return Math.max(
            this.query(start, mid, left, right, 2 * node + 1),
            this.query(mid + 1, end, left, right, 2 * node + 2)
        );
    }

    update(start, end, index, value, node) {
        if (start === end) {
            this.tree[node] = value; // Update the value
            return;
        }

        const mid = Math.floor((start + end) / 2);
        if (index <= mid) {
            this.update(start, mid, index, value, 2 * node + 1);
        } else {
            this.update(mid + 1, end, index, value, 2 * node + 2);
        }

        this.tree[node] = Math.max(
            this.tree[2 * node + 1],
            this.tree[2 * node + 2]
        );
    }
}
```

```

function lengthOfLISSegmentTree(nums) {
    const sorted = [...new Set(nums)].sort((a, b) => a - b);
    const rankMap = new Map(sorted.map((val, idx) => [val, idx]));
    const segmentTree = new SegmentTree(sorted.length);

    let maxLength = 0;
    for (const num of nums) {
        const rank = rankMap.get(num);
        const currentLength =
            segmentTree.query(0, sorted.length - 1, 0, rank - 1, 0) + 1;
        segmentTree.update(0, sorted.length - 1, rank, currentLength,
0);
        maxLength = Math.max(maxLength, currentLength);
    }

    return maxLength;
}

// Test
console.log(lengthOfLISSegmentTree([10, 9, 2, 5, 3, 7, 101, 18]));
// Output: 4
console.log(lengthOfLISSegmentTree([0, 1, 0, 3, 2, 3])); // Output:
4
console.log(lengthOfLISSegmentTree([7, 7, 7, 7])); // Output: 1

```

### Explanation:

1. Use a Segment Tree to query and update the maximum LIS length efficiently.
2. Compress the input array to map values to ranks for Segment Tree indexing.

### Time Complexity:

- **O(n log n)**: Each update and query operation takes logarithmic time.

### Space Complexity:

- **O(n)**: Space for the Segment Tree.

## Longest Increasing Subsequence - Count Number of LIS

### 379. Problem: Find the number of longest increasing subsequences in the array.

### Solution:

```

function countLIS(nums) {
    const n = nums.length;
    const dp = new Array(n).fill(1);

```

```

const count = new Array(n).fill(1);

let maxLength = 1;

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

return count.reduce((sum, c, i) => (dp[i] === maxLength ? sum + c : sum), 0);
}

// Test
console.log(countLIS([1, 3, 5, 4, 7])); // Output: 2
console.log(countLIS([2, 2, 2, 2, 2])); // Output: 5
console.log(countLIS([10, 9, 2, 5, 3, 7, 101, 18])); // Output: 1

```

### Explanation :

1. Use DP to compute the length of the LIS and the count of LIS for each index.
2. Define:
  - o  $dp[i]$  as the length of the LIS ending at index  $i$ .
  - o  $count[i]$  as the number of LIS ending at index  $i$ .
3. Transition:
  - o For each  $i$  and  $j$  ( $j < i$ ):
    - If  $nums[j] < nums[i]$ :
      - If  $dp[j] + 1 > dp[i]$ , update  $dp[i]$  and set  $count[i] = count[j]$ .
      - If  $dp[j] + 1 === dp[i]$ , add  $count[j]$  to  $count[i]$ .
4. Sum the counts for indices where  $dp[i]$  equals the maximum LIS length.

### Time Complexity:

- **$O(n^2)$ :** Two nested loops for DP computation.

### Space Complexity:

- **O(n)**: Space for `dp` and `count` arrays.

### Longest Increasing Subsequence - Using Patience Sorting

**380. Problem: Revisit the binary search solution with a focus on the mechanics of Patience Sorting.**

**Solution:**

```
function lengthOfLISPatience(nums) {
  const piles = [];

  for (const num of nums) {
    let left = 0,
        right = piles.length;
    while (left < right) {
      const mid = Math.floor((left + right) / 2);
      if (piles[mid] < num) {
        left = mid + 1;
      } else {
        right = mid;
      }
    }
    if (left < piles.length) {
      piles[left] = num;
    } else {
      piles.push(num);
    }
  }

  return piles.length;
}

// Test
console.log(lengthOfLISPatience([10, 9, 2, 5, 3, 7, 101, 18])); // Output: 4
console.log(lengthOfLISPatience([0, 1, 0, 3, 2, 3])); // Output: 4
console.log(lengthOfLISPatience([7, 7, 7, 7])); // Output: 1
```

**Explanation:**

1. Treat the LIS problem as a process of building piles where each pile represents a potential subsequence.
2. Use binary search to efficiently find the pile where the current number can be placed.

**Time Complexity:**

- **O(n log n)**: Binary search for each element.

## Space Complexity:

- **O(n)**: Space for the piles array.

## Longest Common Subsequence - Using Dynamic Programming

**381. Problem:** Given two strings `text1` and `text2`, return the length of their longest common subsequence. If there is no common subsequence, return 0.

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace".
```

**Solution:**

```
function longestCommonSubsequence(text1, text2) {
    const m = text1.length,
        n = text2.length;
    const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

    for (let i = 1; i <= m; i++) {
        for (let j = 1; j <= n; j++) {
            if (text1[i - 1] === text2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[m][n];
}

// Test
console.log(longestCommonSubsequence("abcde", "ace")); // Output: 3
console.log(longestCommonSubsequence("abc", "abc")); // Output: 3
console.log(longestCommonSubsequence("abc", "def")); // Output: 0
```

## Explanation :

1. Use a **Dynamic Programming (DP)** table.
2. Define `dp[i][j]` as the length of the LCS of the first `i` characters of `text1` and the first `j` characters of `text2`.
3. Transition:
  - o If `text1[i-1] === text2[j-1]`, then `dp[i][j] = dp[i-1][j-1] + 1`.
  - o Otherwise, `dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1])`.

4. Return  $dp[m][n]$  where  $m$  and  $n$  are the lengths of  $text1$  and  $text2$ .

### Time Complexity:

- $O(m * n)$ : Two nested loops for the DP table.

### Space Complexity:

- $O(m * n)$ : Space for the DP table.

## Longest Common Subsequence - Space Optimized

### 382. Problem: Reduce the space complexity of the LCS solution.

#### Solution:

```
function longestCommonSubsequenceOptimized(text1, text2) {  
    const m = text1.length,  
        n = text2.length;  
    let prev = new Array(n + 1).fill(0);  
  
    for (let i = 1; i <= m; i++) {  
        const curr = new Array(n + 1).fill(0);  
        for (let j = 1; j <= n; j++) {  
            if (text1[i - 1] === text2[j - 1]) {  
                curr[j] = prev[j - 1] + 1;  
            } else {  
                curr[j] = Math.max(prev[j], curr[j - 1]);  
            }  
        }  
        prev = curr;  
    }  
  
    return prev[n];  
}  
  
// Test  
console.log(longestCommonSubsequenceOptimized("abcde", "ace")); //  
Output: 3  
console.log(longestCommonSubsequenceOptimized("abc", "abc")); //  
Output: 3  
console.log(longestCommonSubsequenceOptimized("abc", "def")); //  
Output: 0
```

### Explanation:

1. Use  $prev$  to store the results of the previous row.

2. Update `curr` for the current row, then swap `prev` and `curr`.

### Time Complexity:

- **O(m \* n)**: Same as the standard DP solution.

### Space Complexity:

- **O(n)**: Space for two 1D arrays.

## Longest Common Subsequence - Recursive with Memoization

### 383. Problem: Use a top-down recursive approach with memoization.

#### Solution:

```
function longestCommonSubsequenceRecursive(text1, text2) {
  const memo = new Map();

  function helper(i, j) {
    if (i < 0 || j < 0) return 0;

    const key = `${i},${j}`;
    if (memo.has(key)) return memo.get(key);

    if (text1[i] === text2[j]) {
      memo.set(key, 1 + helper(i - 1, j - 1));
    } else {
      memo.set(key, Math.max(helper(i - 1, j), helper(i, j - 1)));
    }

    return memo.get(key);
  }

  return helper(text1.length - 1, text2.length - 1);
}

// Test
console.log(longestCommonSubsequenceRecursive("abcde", "ace")); // Output: 3
console.log(longestCommonSubsequenceRecursive("abc", "abc")); // Output: 3
console.log(longestCommonSubsequenceRecursive("abc", "def")); // Output: 0
```

#### Explanation :

1. Define a recursive function `helper(i, j)`:

- o If  $\text{text1}[i] == \text{text2}[j]$ , add 1 to the result of  $\text{helper}(i-1, j-1)$ .
  - o Otherwise, return the maximum of  $\text{helper}(i-1, j)$  and  $\text{helper}(i, j-1)$ .
2. Use a memoization table to store results for  $(i, j)$ .

### Time Complexity:

- $O(m * n)$ : Each state  $(i, j)$  is computed once.

### Space Complexity:

- $O(m * n)$ : Space for memoization.

## Longest Common Subsequence - Print the Subsequence

### 384. Problem: Return the actual longest common subsequence.

#### Solution:

```
function printLCS(text1, text2) {
  const m = text1.length,
    n = text2.length;
  const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (text1[i - 1] === text2[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1] + 1;
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }

  let i = m,
    j = n;
  const lcs = [];
  while (i > 0 && j > 0) {
    if (text1[i - 1] === text2[j - 1]) {
      lcs.unshift(text1[i - 1]);
      i--;
      j--;
    } else if (dp[i - 1][j] > dp[i][j - 1]) {
      i--;
    } else {
      j--;
    }
  }
}
```

```

    }

    return lcs.join("");
}

// Test
console.log(printLCS("abcde", "ace")); // Output: "ace"
console.log(printLCS("abc", "abc")); // Output: "abc"
console.log(printLCS("abc", "def")); // Output: ""

```

### Longest Common Subsequence - Print All Subsequences

**385. Problem:** **Find and return all the longest common subsequences between two strings.**

**Solution:**

```

function findAllLCS(text1, text2) {
  const m = text1.length,
    n = text2.length;
  const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

  // Build the DP table
  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (text1[i - 1] === text2[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1] + 1;
      } else {
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }

  const result = new Set();

  // Helper function to backtrack and find all LCSs
  function backtrack(i, j, path) {
    if (i === 0 || j === 0) {
      result.add(path);
      return;
    }

    if (text1[i - 1] === text2[j - 1]) {
      backtrack(i - 1, j - 1, text1[i - 1] + path);
    } else {
      if (dp[i - 1][j] === dp[i][j]) backtrack(i - 1, j, path);
    }
  }
}

```

```

        if (dp[i][j - 1] === dp[i][j]) backtrack(i, j - 1, path);
    }

    backtrack(m, n, "");
    return Array.from(result);
}

// Test
console.log(findAllLCS("abcde", "ace")); // Output: ["ace"]
console.log(findAllLCS("abc", "abc")); // Output: ["abc"]
console.log(findAllLCS("abc", "def")); // Output: []

```

### Explanation:

1. Use the DP table to compute the LCS length.
2. Backtrack from  $dp[m][n]$  to generate all possible LCSSs by exploring all valid paths.

### Time Complexity:

- $O(2^{m+n})$ : Backtracking can generate multiple paths in the worst case.

### Space Complexity:

- $O(m * n)$ : Space for the DP table.

## Longest Common Subsequence - Variations with Deletions

**386. Problem: Modify the LCS problem to find the minimum number of deletions required to make the two strings identical.**

### Solution:

```

function minDeletionsForLCS(text1, text2) {
    const m = text1.length,
        n = text2.length;
    const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

    // Build the DP table
    for (let i = 1; i <= m; i++) {
        for (let j = 1; j <= n; j++) {
            if (text1[i - 1] === text2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
}

```

```

        }
    }

    const lcsLength = dp[m][n];
    return m - lcsLength + (n - lcsLength);
}

// Test
console.log(minDeletionsForLCS("abcde", "ace")); // Output: 2
console.log(minDeletionsForLCS("sea", "eat")); // Output: 2
console.log(minDeletionsForLCS("abcdef", "ghijk")); // Output: 10

```

### Explanation:

1. Compute the LCS length using the DP approach.
2. Calculate the minimum deletions as  $(\text{text1.length} - \text{LCS}) + (\text{text2.length} - \text{LCS})$ .

### Time Complexity:

- $O(m * n)$ : Standard DP table computation.

### Space Complexity:

- $O(m * n)$ : Space for the DP table.

## Longest Common Subsequence - Shortest Common Supersequence

**387. Problem:** **Find the shortest common supersequence (SCS) of two strings. The SCS is the shortest string that contains both strings as subsequences.**

### Solution:

```

function shortestCommonSupersequence(text1, text2) {
    const m = text1.length,
        n = text2.length;
    const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

    // Build the DP table
    for (let i = 1; i <= m; i++) {
        for (let j = 1; j <= n; j++) {
            if (text1[i - 1] === text2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    let result = '';
    let i = m, j = n;
    while (i > 0 && j > 0) {
        if (text1[i - 1] === text2[j - 1]) {
            result += text1[i - 1];
            i--;
            j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            result += text1[i - 1];
            i--;
        } else {
            result += text2[j - 1];
            j--;
        }
    }

    if (i > 0) {
        result += text1.substring(i);
    } else if (j > 0) {
        result += text2.substring(j);
    }

    return result;
}

```

```

        }
    }

let i = m,
    j = n;
const scs = [];

while (i > 0 && j > 0) {
    if (text1[i - 1] === text2[j - 1]) {
        scs.unshift(text1[i - 1]);
        i--;
        j--;
    } else if (dp[i - 1][j] > dp[i][j - 1]) {
        scs.unshift(text1[i - 1]);
        i--;
    } else {
        scs.unshift(text2[j - 1]);
        j--;
    }
}

while (i > 0) {
    scs.unshift(text1[i - 1]);
    i--;
}

while (j > 0) {
    scs.unshift(text2[j - 1]);
    j--;
}

return scs.join("");
}

// Test
console.log(shortestCommonSupersequence("abcde", "ace")); // Output:
"abcde"
console.log(shortestCommonSupersequence("sea", "eat")); // Output:
"seaat"
console.log(shortestCommonSupersequence("abcdef", "ghijk")); // Output:
"abcdefghijk"

```

### Explanation:

1. Use the LCS to guide the construction of the SCS.
2. Add non-matching characters from both strings to the result.

### Time Complexity:

- $O(m * n)$ : LCS computation and reconstruction.

### Space Complexity:

- $O(m * n)$ : Space for the DP table.

## Longest Common Subsequence - Longest Palindromic Subsequence

**388. Problem:** Given a string  $s$ , find the length of its longest palindromic subsequence.

### Solution:

```
function longestPalindromicSubsequence(s) {  
    const reverseS = s.split("").reverse().join("");  
    const m = s.length,  
        n = reverseS.length;  
    const dp = Array.from({ length: m + 1 }, () => Array(n +  
1).fill(0));  
  
    // Build the DP table  
    for (let i = 1; i <= m; i++) {  
        for (let j = 1; j <= n; j++) {  
            if (s[i - 1] === reverseS[j - 1]) {  
                dp[i][j] = dp[i - 1][j - 1] + 1;  
            } else {  
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);  
            }  
        }  
    }  
  
    return dp[m][n];  
}  
  
// Test  
console.log(longestPalindromicSubsequence("bbbab")); // Output: 4  
console.log(longestPalindromicSubsequence("cbbd")); // Output: 2  
console.log(longestPalindromicSubsequence("abc")); // Output: 1
```

### Explanation:

1. Reverse the string  $s$  and compute the LCS between  $s$  and its reverse.
2. The result will be the length of the longest palindromic subsequence.

### Time Complexity:

- $O(n^2)$ : LCS computation for strings of length  $n$ .

### Space Complexity:

- $O(n^2)$ : Space for the DP table.

### Longest Common Subsequence - Longest Repeated Subsequence

**389. Problem:** Find the longest subsequence that appears at least twice in the given string  $s$ .

### Solution:

```
function longestRepeatedSubsequence(s) {  
    const m = s.length;  
    const dp = Array.from({ length: m + 1 }, () => Array(m +  
1).fill(0));  
  
    // Build the DP table  
    for (let i = 1; i <= m; i++) {  
        for (let j = 1; j <= m; j++) {  
            if (s[i - 1] === s[j - 1] && i !== j) {  
                dp[i][j] = dp[i - 1][j - 1] + 1;  
            } else {  
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);  
            }  
        }  
    }  
  
    return dp[m][m];  
}  
  
// Test  
console.log(longestRepeatedSubsequence("aab")); // Output: 1  
console.log(longestRepeatedSubsequence("aabb")); // Output: 2  
console.log(longestRepeatedSubsequence("axxxy")); // Output: 2
```

### Explanation:

1. Use a modified LCS approach where the indices of matching characters must be different.
2. The result will be the length of the longest repeated subsequence.

### Time Complexity:

- $O(n^2)$ : LCS computation for the string with itself.

### Space Complexity:

- $O(n^2)$ : Space for the DP table.

## Longest Common Subsequence - Minimum Insertions to Make a String Palindrome

**390. Problem:** Given a string  $s$ , find the minimum number of insertions needed to make it a palindrome.

**Solution:**

```
function minInsertionsToPalindrome(s) {  
    const lpsLength = longestPalindromicSubsequence(s); // Reuse the  
    previous function  
    return s.length - lpsLength;  
}  
  
// Test  
console.log(minInsertionsToPalindrome("mbadm")); // Output: 2  
console.log(minInsertionsToPalindrome("zzazz")); // Output: 0  
console.log(minInsertionsToPalindrome("leetcode")); // Output: 5
```

**Explanation:**

1. Compute the LPS of  $s$  using the LCS approach.
2. Subtract the length of the LPS from the length of  $s$  to get the minimum insertions required.

**Time Complexity:**

- $O(n^2)$ : LCS computation for  $s$  and its reverse.

**Space Complexity:**

- $O(n^2)$ : Space for the DP table.

## Breaking Words into Meaningful Segments - Check If a Word Can Be Segmented

**391. Problem:**

Given a string  $s$  and a dictionary of strings  $wordDict$ , return `true` if  $s$  can be segmented into a space-separated sequence of one or more dictionary words.

```
Input: s = "leetcode", wordDict = ["leet", "code"]  
Output: true  
Explanation: "leetcode" can be segmented as "leet code".  
  
Input: s = "applepenapple", wordDict = ["apple", "pen"]  
Output: true  
Explanation: "applepenapple" can be segmented as "apple pen apple".
```

### Solution:

```
function wordBreak(s, wordDict) {  
    const wordSet = new Set(wordDict);  
    const dp = new Array(s.length + 1).fill(false);  
    dp[0] = true; // Empty string is always valid  
  
    for (let i = 1; i <= s.length; i++) {  
        for (let j = 0; j < i; j++) {  
            if (dp[j] && wordSet.has(s.substring(j, i))) {  
                dp[i] = true;  
                break;  
            }  
        }  
    }  
  
    return dp[s.length];  
}  
  
// Test  
console.log(wordBreak("leetcode", ["leet", "code"])); // Output:  
true  
console.log(wordBreak("applepenapple", ["apple", "pen"])); //  
Output: true  
console.log(wordBreak("catsandog", ["cats", "dog", "sand", "and",  
"cat"])); // Output: false
```

### Explanation:

1. Iterate through the string  $s$  and check if each substring can be formed using dictionary words.
2. Use the  $dp$  array to store results for all substrings up to index  $i$ .

### Time Complexity:

- $O(n^2)$ : Outer loop runs  $n$  times, and the inner loop checks all substrings.

### Space Complexity:

- $O(n)$ : Space for the  $dp$  array.

## Breaking Words into Meaningful Segments - Return All Possible Segmentations

### 392. Problem:

**Given a string  $s$  and a dictionary  $wordDict$ , return all possible segmentations of  $s$  where each word is in  $wordDict$ .**

### Solution:

```

function wordBreakAll(s, wordDict) {
  const wordSet = new Set(wordDict);
  const memo = new Map();

  function backtrack(start) {
    if (memo.has(start)) return memo.get(start);
    if (start === s.length) return [""];

    const results = [];
    for (let end = start + 1; end <= s.length; end++) {
      const word = s.substring(start, end);
      if (wordSet.has(word)) {
        const nextSegments = backtrack(end);
        for (const segment of nextSegments) {
          results.push(word + (segment ? " " + segment : ""));
        }
      }
    }

    memo.set(start, results);
    return results;
  }

  return backtrack(0);
}

// Test
console.log(wordBreakAll("catsanddog", ["cat", "cats", "and",
"sand", "dog"]));
// Output: ["cat sand dog", "cats and dog"]
console.log(
  wordBreakAll("pineapplepenapple", [
    "apple",
    "pen",
    "applepen",
    "pine",
    "pineapple",
  ])
);
// Output: ["pine apple pen apple", "pineapple pen apple", "pine
applepen apple"]

```

### Explanation:

1. Use recursion to explore all valid segmentations.
2. Memoize results for each starting index to avoid redundant computations.

### Time Complexity:

- $O(n^2 + 2^n)$ : Substring checks and exponential recursion.

### Space Complexity:

- $O(n)$ : Space for recursion stack and memoization.

## Breaking Words into Meaningful Segments - Minimum Number of Segments

### 393. Problem:

**Find the minimum number of words needed to segment the string  $s$  using `wordDict`. Return  $-1$  if segmentation is not possible.**

### Solution:

```
function minWordBreak(s, wordDict) {
  const wordSet = new Set(wordDict);
  const dp = new Array(s.length + 1).fill(Infinity);
  dp[0] = 0;

  for (let i = 1; i <= s.length; i++) {
    for (let j = 0; j < i; j++) {
      if (dp[j] !== Infinity && wordSet.has(s.substring(j, i))) {
        dp[i] = Math.min(dp[i], dp[j] + 1);
      }
    }
  }

  return dp[s.length] === Infinity ? -1 : dp[s.length];
}

// Test
console.log(minWordBreak("leetcode", ["leet", "code"])); // Output: 2
console.log(minWordBreak("applepenapple", ["apple", "pen"])); // Output: 3
console.log(minWordBreak("catsandog", ["cats", "dog", "sand", "and", "cat"])); // Output: -1
```

## Breaking Words into Meaningful Segments - Boolean Check with Trie

### 394. Problem: Optimize the Breaking Words into Meaningful Segments using a Trie for efficient prefix checking.

### Solution:

```
class TrieNode {
  constructor() {
    this.children = {};
    this.isEnd = false;
```

```

    }

}

function buildTrie(wordDict) {
  const root = new TrieNode();
  for (const word of wordDict) {
    let node = root;
    for (const char of word) {
      if (!node.children[char]) node.children[char] = new
TrieNode();
      node = node.children[char];
    }
    node.isEnd = true;
  }
  return root;
}

function wordBreakWithTrie(s, wordDict) {
  const root = buildTrie(wordDict);
  const dp = new Array(s.length + 1).fill(false);
  dp[0] = true;

  for (let i = 1; i <= s.length; i++) {
    let node = root;
    for (let j = i - 1; j >= 0; j--) {
      const char = s[j];
      if (!node.children[char]) break;
      node = node.children[char];
      if (node.isEnd && dp[j]) {
        dp[i] = true;
        break;
      }
    }
  }

  return dp[s.length];
}

// Test
console.log(wordBreakWithTrie("leetcode", ["leet", "code"])); // Output: true
console.log(wordBreakWithTrie("applepenapple", ["apple", "pen"])); // Output: true
console.log(
  wordBreakWithTrie("catsandog", ["cats", "dog", "sand", "and",
"cat"]))
); // Output: false

```

### **Explanation:**

1. Build a Trie to store the words in `wordDict`.
2. Use the Trie to validate substrings during DP computation.

### **Time Complexity:**

- **O(n<sup>2</sup>)**: DP computation with substring checks in the Trie.

### **Space Complexity:**

- **O(n)**: Space for the Trie and DP array.

## **Breaking Words into Meaningful Segments - BFS Approach**

**395. Problem: Determine if the string `s` can be segmented into one or more words from `wordDict` using Breadth-First Search (BFS).**

### **Solution:**

```
function wordBreakBFS(s, wordDict) {  
    const wordSet = new Set(wordDict);  
    const queue = [0];  
    const visited = new Set();  
  
    while (queue.length > 0) {  
        const start = queue.shift();  
        if (visited.has(start)) continue;  
        visited.add(start);  
  
        for (let end = start + 1; end <= s.length; end++) {  
            if (wordSet.has(s.substring(start, end))) {  
                if (end === s.length) return true;  
                queue.push(end);  
            }  
        }  
    }  
  
    return false;  
}  
  
// Test  
console.log(wordBreakBFS("leetcode", ["leet", "code"])); // Output:  
true  
console.log(wordBreakBFS("applepenapple", ["apple", "pen"])); //  
Output: true  
console.log(wordBreakBFS("catsandog", ["cats", "dog", "sand", "and",  
"cat"])); // Output: false
```

### Explanation:

1. Use BFS to explore all possible segmentations.
2. Check each substring against the `wordSet` and continue exploring until the end of the string.

### Time Complexity:

- $O(n^2)$ : BFS traversal and substring checks.

### Space Complexity:

- $O(n)$ : Space for the queue and visited set.

## Breaking Words into Meaningful Segments - DFS with Memoization

### 396. Problem:

**Check if the string `s` can be segmented into one or more words from `wordDict` using Depth-First Search (DFS) with memoization.**

### Solution:

```
function wordBreakDFS(s, wordDict) {  
    const wordSet = new Set(wordDict);  
    const memo = new Map();  
  
    function dfs(start) {  
        if (start === s.length) return true;  
        if (memo.has(start)) return memo.get(start);  
  
        for (let end = start + 1; end <= s.length; end++) {  
            if (wordSet.has(s.substring(start, end)) && dfs(end)) {  
                memo.set(start, true);  
                return true;  
            }  
        }  
  
        memo.set(start, false);  
        return false;  
    }  
  
    return dfs(0);  
}  
  
// Test  
console.log(wordBreakDFS("leetcode", ["leet", "code"])); // Output:  
true  
console.log(wordBreakDFS("applepenapple", ["apple", "pen"])); //  
Output: true
```

```
console.log(wordBreakDFS("catsandog", ["cats", "dog", "sand", "and", "cat"])); // Output: false
```

### Explanation:

1. Use DFS to explore all possible segmentations.
2. Memoize results for each starting index to avoid redundant computations.

### Time Complexity:

- $O(n^2)$ : Recursive traversal with substring checks.

### Space Complexity:

- $O(n)$ : Space for the memoization object and recursion stack.

## Breaking Words into Meaningful Segments - Maximum Number of Segments

**397. Problem: Find the maximum number of segments that can be formed from the string s using words from wordDict.**

### Solution:

```
function maxWordBreakSegments(s, wordDict) {  
    const wordSet = new Set(wordDict);  
    const dp = new Array(s.length + 1).fill(-Infinity);  
    dp[0] = 0;  
  
    for (let i = 1; i <= s.length; i++) {  
        for (let j = 0; j < i; j++) {  
            if (dp[j] !== -Infinity && wordSet.has(s.substring(j, i))) {  
                dp[i] = Math.max(dp[i], dp[j] + 1);  
            }  
        }  
    }  
  
    return dp[s.length] === -Infinity ? -1 : dp[s.length];  
}  
  
// Test  
console.log(maxWordBreakSegments("leetcode", ["leet", "code"])); // Output: 2  
console.log(maxWordBreakSegments("applepenapple", ["apple", "pen"])); // Output: 3  
console.log(  
    maxWordBreakSegments("catsandog", ["cats", "dog", "sand", "and", "cat"])  
); // Output: -1
```

## **Explanation:**

1. Use a DP approach to compute the maximum number of segments.
2. Update  $dp[i]$  for each valid substring.

## **Time Complexity:**

- $O(n^2)$ : DP computation with substring checks.

## **Space Complexity:**

- $O(n)$ : Space for the DP array.

## **Breaking Words into Meaningful Segments - Find All Unique Segmentations**

**398. Problem: Return all unique segmentations of the string  $s$  using `wordDict` without duplicates.**

## **Solution:**

```
function uniqueWordBreaks(s, wordDict) {  
    const wordSet = new Set(wordDict);  
    const result = new Set();  
  
    function backtrack(start, path) {  
        if (start === s.length) {  
            result.add(path.join(" "));  
            return;  
        }  
  
        for (let end = start + 1; end <= s.length; end++) {  
            const word = s.substring(start, end);  
            if (wordSet.has(word)) {  
                path.push(word);  
                backtrack(end, path);  
                path.pop();  
            }  
        }  
    }  
  
    backtrack(0, []);  
    return Array.from(result);  
}  
  
// Test  
console.log(
```

```

uniqueWordBreaks("catsanddog", ["cat", "cats", "and", "sand",
"dog"])
);
// Output: ["cat sand dog", "cats and dog"]
console.log(
  uniqueWordBreaks("pineapplepenapple", [
    "apple",
    "pen",
    "applepen",
    "pine",
    "pineapple",
  ])
);
// Output: ["pine apple pen apple", "pineapple pen apple", "pine
applepen apple"]

```

### Explanation:

1. Use backtracking to explore all possible segmentations.
2. Store each unique segmentation in a set to avoid duplicates.

### Time Complexity:

- $O(n^2 + 2^n)$ : Backtracking and substring checks.

### Space Complexity:

- $O(n)$ : Space for the recursion stack.

## Breaking Words into Meaningful Segments - Find Longest Segmentable Substring

**399. Problem:** Given a string  $s$  and a dictionary  $\text{wordDict}$ , find the longest segmentable substring of  $s$  such that it can be segmented using  $\text{wordDict}$ . If no such substring exists, return an empty string.

### Solution:

```

function longestSegmentableSubstring(s, wordDict) {
  const wordSet = new Set(wordDict);
  const dp = new Array(s.length + 1).fill(false);
  dp[0] = true; // Empty string is always valid

  let longestSubstring = "";

  for (let i = 1; i <= s.length; i++) {
    for (let j = 0; j < i; j++) {
      if (dp[j] && wordSet.has(s.substring(j, i))) {
        dp[i] = true;
        longestSubstring = s.substring(j, i);
      }
    }
  }
}

```

```

        if (i - j > longestSubstring.length) {
            longestSubstring = s.substring(j, i);
        }
        break;
    }
}

return longestSubstring;
}

// Test
console.log(longestSegmentableSubstring("leetcode", ["leet", "code"])); // Output: "leetcode"
console.log(
    longestSegmentableSubstring("catsanddog", [
        "cats",
        "and",
        "sand",
        "dog",
        "cat",
    ])
); // Output: "catsanddog"
console.log(longestSegmentableSubstring("applepen", ["apple", "pen", "app"])); // Output: "apple"

```

### Explanation:

1. Use a DP table to track whether a substring can be segmented using wordDict.
2. Update the longest valid substring whenever a valid segmentation is found.

### Time Complexity:

- $O(n^2)$ : DP computation with substring checks.

### Space Complexity:

- $O(n)$ : Space for the DP array.

## Breaking Words into Meaningful Segments - Count All Segmentations

**400. Problem:** Count all possible segmentations of the string  $s$  using words from  $wordDict$ .

### Solution:

```
function countWordBreaks(s, wordDict) {
```

```

const wordSet = new Set(wordDict);
const dp = new Array(s.length + 1).fill(0);
dp[0] = 1; // There is one way to segment an empty string

for (let i = 1; i <= s.length; i++) {
    for (let j = 0; j < i; j++) {
        if (dp[j] > 0 && wordSet.has(s.substring(j, i))) {
            dp[i] += dp[j];
        }
    }
}

return dp[s.length];
}

// Test
console.log(
    countWordBreaks("catsanddog", ["cat", "cats", "and", "sand",
    "dog"]))
); // Output: 2
console.log(countWordBreaks("applepenapple", ["apple", "pen"])); // Output: 1
console.log(
    countWordBreaks("pineapplepenapple", [
        "apple",
        "pen",
        "applepen",
        "pine",
        "pineapple",
    ]))
); // Output: 3

```

### Explanation:

1. Use a DP array to track the count of ways to segment substrings.
2. Add counts from all valid segmentations for each index.

### Time Complexity:

- **O(n<sup>2</sup>):** DP computation with substring checks.

### Space Complexity:

- **O(n):** Space for the DP array.

## Finding the Best Combination Selections - Find All Combinations

### 401. Problem:

Given an array of distinct integers `candidates` and a target integer `target`, return all unique combinations of `candidates` where the chosen numbers sum to `target`. You may reuse the same element multiple times.

#### Example:

```
Input: (candidates = [2, 3, 6, 7]), (target = 7);
Output: [[2, 2, 3], [7]];
```

```
Input: (candidates = [2, 3, 5]), (target = 8);
Output: [
  [2, 2, 2, 2],
  [2, 3, 3],
  [3, 5],
];
```

#### Solution:

```
function combinationSum(candidates, target) {
  const result = [];

  function backtrack(remaining, start, combination) {
    if (remaining === 0) {
      result.push([...combination]);
      return;
    }

    for (let i = start; i < candidates.length; i++) {
      if (candidates[i] <= remaining) {
        combination.push(candidates[i]);
        backtrack(remaining - candidates[i], i, combination); // Allow reuse
        combination.pop();
      }
    }
  }

  backtrack(target, 0, []);
  return result;
}

// Test
console.log(combinationSum([2, 3, 6, 7], 7)); // Output: [[2, 2, 3], [7]]
console.log(combinationSum([2, 3, 5], 8)); // Output: [[2, 2, 2, 2], [2, 3, 3], [3, 5]]
```

### **Explanation:**

1. Start with an empty combination and explore all possibilities recursively.
2. Subtract the current candidate from the target and recurse with the same index.

### **Time Complexity:**

- **O(2^n):** Exponential due to recursive exploration.

### **Space Complexity:**

- **O(n):** Space for the recursion stack.

## **Finding the Best Combination Selections - Return Count of Combinations**

**402. Problem: Count the total number of unique combinations where the numbers sum to the target.**

### **Solution:**

```
function combinationSumCount(candidates, target) {  
    let count = 0;  
  
    function backtrack(remaining, start) {  
        if (remaining === 0) {  
            count++;  
            return;  
        }  
  
        for (let i = start; i < candidates.length; i++) {  
            if (candidates[i] <= remaining) {  
                backtrack(remaining - candidates[i], i); // Allow reuse  
            }  
        }  
    }  
  
    backtrack(target, 0);  
    return count;  
}  
  
// Test  
console.log(combinationSumCount([2, 3, 6, 7], 7)); // Output: 2  
console.log(combinationSumCount([2, 3, 5], 8)); // Output: 3
```

### **Explanation:**

- Instead of storing combinations, increment the count whenever a valid combination is found.
- Explore all possibilities as in the standard backtracking approach.

### Time Complexity:

- $O(2^n)$ : Exponential due to recursive exploration.

### Space Complexity:

- $O(n)$ : Space for the recursion stack.

## Finding the Best Combination Selections - Using Dynamic Programming

**403. Problem: Find all unique combinations where the numbers sum to the target using Dynamic Programming (DP).**

### Solution:

```
function combinationSumDP(candidates, target) {
  const dp = Array.from({ length: target + 1 }, () => []);
  dp[0] = [[]]; // Base case: one way to make 0

  for (const num of candidates) {
    for (let i = num; i <= target; i++) {
      for (const combination of dp[i - num]) {
        dp[i].push([...combination, num]);
      }
    }
  }

  return dp[target];
}

// Test
console.log(combinationSumDP([2, 3, 6, 7], 7)); // Output: [[2, 2, 3], [7]]
console.log(combinationSumDP([2, 3, 5], 8)); // Output: [[2, 2, 2, 2], [2, 3, 3], [3, 5]]
```

### Explanation:

- For each candidate, update all possible sums in `dp`.
- Append the current candidate to each valid combination.

### Time Complexity:

- $O(\text{target} * n)$ : DP array updates for each candidate.

## Space Complexity:

- **O(target \* m)**: Space for the DP array, where  $m$  is the average number of combinations.

## Finding the Best Combination Selections - Allow Unique Candidates Only

**404. Problem: Modify the problem so that each candidate can be used at most once.**

### Solution:

```
function combinationSumUnique(candidates, target) {
    const result = [];

    function backtrack(remaining, start, combination) {
        if (remaining === 0) {
            result.push([...combination]);
            return;
        }

        for (let i = start; i < candidates.length; i++) {
            if (candidates[i] <= remaining) {
                combination.push(candidates[i]);
                backtrack(remaining - candidates[i], i + 1, combination); // Move to
the next index
                combination.pop();
            }
        }
    }

    backtrack(target, 0, []);
    return result;
}

// Test
console.log(combinationSumUnique([2, 3, 6, 7], 7)); // Output: [[7]]
console.log(combinationSumUnique([2, 3, 5], 8)); // Output: [[3, 5]]
```

### Explanation:

1. Increment the index in recursive calls to ensure each candidate is used only once.
2. Explore all valid combinations recursively.

## Time Complexity:

- **O(2^n)**: Exponential due to recursive exploration.

## Space Complexity:

- **O(n)**: Space for the recursion stack.

## Finding the Best Combination Selections - Minimum Number of Combinations

### 405. Problem:

Find the minimum number of elements required to form the target using the given candidates. If it's not possible, return -1.

### Solution:

```
function minCombinationSum(candidates, target) {
    const dp = new Array(target + 1).fill(Infinity);
    dp[0] = 0; // Base case: 0 elements needed to make sum 0

    for (const num of candidates) {
        for (let i = num; i <= target; i++) {
            dp[i] = Math.min(dp[i], dp[i - num] + 1);
        }
    }

    return dp[target] === Infinity ? -1 : dp[target];
}

// Test
console.log(minCombinationSum([2, 3, 6, 7], 7)); // Output: 1 (7)
console.log(minCombinationSum([2, 3, 5], 8)); // Output: 2 (3 + 5)
console.log(minCombinationSum([5, 10], 3)); // Output: -1
```

### Explanation:

1. Use the DP approach to track the minimum elements needed for each sum.
2. Return -1 if `dp[target]` remains Infinity.

### Time Complexity:

- **O(target \* n)**: DP updates for each candidate.

### Space Complexity:

- **O(target)**: Space for the DP array.

## Finding the Best Combination Selections - Maximum Number of Combinations

406. Problem: Find the maximum number of combinations that can be formed to achieve the target using the given candidates.

### Solution:

```
function maxCombinationSum(candidates, target) {  
    const dp = new Array(target + 1).fill(0);  
    dp[0] = 1; // Base case: 1 way to make sum 0  
  
    for (const num of candidates) {  
        for (let i = num; i <= target; i++) {  
            dp[i] += dp[i - num];  
        }  
    }  
  
    return dp[target];  
}  
  
// Test  
console.log(maxCombinationSum([2, 3, 6, 7], 7)); // Output: 2 ([2,2,3], [7])  
console.log(maxCombinationSum([2, 3, 5], 8)); // Output: 3 ([2,2,2,2],  
[2,3,3], [3,5])  
console.log(maxCombinationSum([5, 10], 3)); // Output: 0
```

### Explanation:

1. Use a DP array to track the number of ways to form each sum.
2. For each candidate, update all sums greater than or equal to the candidate.

### Time Complexity:

- $O(\text{target} * n)$ : DP updates for each candidate.

### Space Complexity:

- $O(\text{target})$ : Space for the DP array.

## Finding the Best Combination Selections - Allow Repeated Combinations

### 407. Problem:

Return all combinations where repeated use of the same set of candidates is allowed (e.g., [2, 3] and [3, 2] are considered different).

### Solution:

```
function combinationSumWithRepetition(candidates, target) {  
    const result = [];  
  
    function backtrack(remaining, combination) {  
        if (remaining === 0) {  
            result.push([...combination]);  
            return;  
        }  
  
        for (const candidate of candidates) {  
            const newCombination = [...combination, candidate];  
            const newRemaining = remaining - candidate;  
            backtrack(newRemaining, newCombination);  
        }  
    }  
  
    backtrack(target, []);  
    return result;  
}
```

```

    }

    for (const num of candidates) {
        if (num <= remaining) {
            combination.push(num);
            backtrack(remaining - num, combination);
            combination.pop();
        }
    }
}

backtrack(target, []);
return result;
}

// Test
console.log(combinationSumWithRepetition([2, 3], 5)); // Output: [[2, 3], [3, 2]]
console.log(combinationSumWithRepetition([2, 3, 6], 6)); // Output: [[2, 2, 2], [3, 3], [6]]

```

### Explanation:

1. Unlike the standard backtracking approach, explore all permutations by iterating over all candidates.

### Time Complexity:

- **O( $2^n$ ):** Exponential due to recursive exploration.

### Space Complexity:

- **O(n):** Space for the recursion stack.

## Finding the Best Combination Selections - Count Ways with Modulo

**408. Problem:** Find the total number of combinations where the sum equals the target, but return the result modulo  $10^9 + 7$ .

### Solution:

```

function combinationSumModulo(candidates, target) {
    const MOD = 1e9 + 7;
    const dp = new Array(target + 1).fill(0);
    dp[0] = 1; // Base case: 1 way to make sum 0

    for (const num of candidates) {
        for (let i = num; i <= target; i++) {

```

```

        dp[i] = (dp[i] + dp[i - num]) % MOD;
    }

}

return dp[target];
}

// Test
console.log(combinationSumModulo([2, 3, 5], 8)); // Output: 3
console.log(combinationSumModulo([1, 2, 3], 10)); // Output: 14

```

### Explanation:

1. Use modulo operation at each step to ensure the result stays within bounds.
2. The final result will be the count modulo  $10^9 + 7$ .

### Time Complexity:

- $O(\text{target} * n)$ : DP updates for each candidate.

### Space Complexity:

- $O(\text{target})$ : Space for the DP array.

## Finding the Best Combination Selections - Find Combinations with Distinct Candidates

**409. Problem:** **Find all combinations where each candidate can be used only once and candidates are unique in the final output.**

### Solution:

```

function combinationSumDistinct(candidates, target) {
    candidates.sort((a, b) => a - b); // Sort candidates
    const result = [];

    function backtrack(remaining, start, combination) {
        if (remaining === 0) {
            result.push([...combination]);
            return;
        }

        for (let i = start; i < candidates.length; i++) {
            if (i > start && candidates[i] === candidates[i - 1]) continue; // Skip duplicates
            if (candidates[i] <= remaining) {
                combination.push(candidates[i]);
                backtrack(remaining - candidates[i], i + 1, combination);
            }
        }
    }

    backtrack(target, 0, []);
}

```

```

        combination.pop();
    }
}

backtrack(target, 0, []);
return result;
}

// Test
console.log(combinationSumDistinct([10, 1, 2, 7, 6, 1, 5], 8));
// Output: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]
console.log(combinationSumDistinct([2, 5, 2, 1, 2], 5));
// Output: [[1, 2, 2], [5]]

```

### Explanation:

1. Sort the candidates and skip duplicate elements to ensure unique combinations.
2. Use backtracking to explore all valid combinations.

### Time Complexity:

- **O(2^n)**: Exponential due to recursive exploration.

### Space Complexity:

- **O(n)**: Space for the recursion stack.

## Finding the Best Combination Selections - Minimum Finding the Best Combination Selections (Greedy)

**410. Problem: Find a combination of numbers from `candidates` such that their sum is closest to the target without exceeding it.**

### Solution:

```

function closestCombinationSum(candidates, target) {
  candidates.sort((a, b) => b - a);
  const result = [];
  let sum = 0;

  for (const num of candidates) {
    while (sum + num <= target) {
      result.push(num);
      sum += num;
    }
  }
}

```

```

        return sum === target ? result : [];
    }

// Test
console.log(closestCombinationSum([2, 3, 6, 7], 7)); // Output: [7]
console.log(closestCombinationSum([2, 3, 5], 8)); // Output: [5, 3]
console.log(closestCombinationSum([5, 10], 3)); // Output: []

```

### Explanation:

1. Use backtracking to explore all combinations.
2. Maintain a variable `closest` to store the closest sum to the target.
3. Update the result whenever a closer sum is found.

### Time Complexity:

- **O(2^n)**: Exponential due to recursive exploration of combinations.

### Space Complexity:

- **O(n)**: Space for the recursion stack.

## House Robber - Maximum Robbery Amount

**411. Problem:** You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. However, adjacent houses have security systems connected, and robbing two adjacent houses will trigger the alarm.

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob **without robbing two adjacent houses**.

```

Input: nums = [2, 7, 9, 3, 1]
Output: 12
Explanation: Rob house 1 (money = 2) and house 3 (money = 9) for a total of 2 + 9 = 12.

Input: nums = [2, 1, 1, 2]
Output: 4
Explanation: Rob house 1 (money = 2) and house 4 (money = 2) for a total of 2 + 2 = 4.

```

### Solution:

```

function rob(nums) {
    const n = nums.length;
    if (n === 0) return 0;
    if (n === 1) return nums[0];
}

```

```

const dp = new Array(n).fill(0);
dp[0] = nums[0];
dp[1] = Math.max(nums[0], nums[1]);

for (let i = 2; i < n; i++) {
    dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
}

return dp[n - 1];
}

// Test
console.log(rob([2, 7, 9, 3, 1])); // Output: 12
console.log(rob([2, 1, 1, 2])); // Output: 4
console.log(rob([1, 2, 3, 1])); // Output: 4

```

### Explanation:

1. For each house  $i$ , decide whether to rob it or skip it:
  - o Robbing it adds  $\text{nums}[i]$  to the value two houses before ( $\text{dp}[i-2]$ ).
  - o Skipping it takes the value of the previous house ( $\text{dp}[i-1]$ ).
2. Use a DP table to calculate the maximum profit iteratively.

### Time Complexity:

- $O(n)$ : Iterate through the array once.

### Space Complexity:

- $O(n)$ : Space for the DP array.

## House Robber - Space Optimized

**412. Problem: Optimize the space complexity of the solution to  $O(1)$  by eliminating the DP array.**

### Solution:

```

function robOptimized(nums) {
    const n = nums.length;
    if (n === 0) return 0;
    if (n === 1) return nums[0];

    let prev2 = 0,
        prev1 = 0;

    for (const num of nums) {
        const current = Math.max(prev1, prev2 + num);

```

```

        prev2 = prev1;
        prev1 = current;
    }

    return prev1;
}

// Test
console.log(robOptimized([2, 7, 9, 3, 1])); // Output: 12
console.log(robOptimized([2, 1, 1, 2])); // Output: 4
console.log(robOptimized([1, 2, 3, 1])); // Output: 4

```

### Explanation:

1. Use two variables to store the maximum profit for the previous two houses.
2. Update the variables iteratively without using extra space.

### Time Complexity:

- **O(n):** Iterate through the array once.

### Space Complexity:

- **O(1):** Only two variables are used.

## House Robber - Print Robbed Houses

**413. Problem:** In addition to finding the maximum amount of money that can be robbed, return the indices of the houses that should be robbed to achieve the maximum amount.

### Solution:

```

function robWithHouses(nums) {
    const n = nums.length;
    if (n === 0) return { maxAmount: 0, houses: [] };
    if (n === 1) return { maxAmount: nums[0], houses: [0] };

    const dp = new Array(n).fill(0);
    const decision = new Array(n).fill(false);

    dp[0] = nums[0];
    decision[0] = true;

    dp[1] = Math.max(nums[0], nums[1]);
    decision[1] = nums[1] > nums[0];

    for (let i = 2; i < n; i++) {
        if (dp[i - 1] > dp[i - 2] + nums[i]) {
            dp[i] = dp[i - 1];
            decision[i] = false;
        } else {
            dp[i] = dp[i - 2] + nums[i];
            decision[i] = true;
        }
    }
}

```

```

        dp[i] = dp[i - 1];
    } else {
        dp[i] = dp[i - 2] + nums[i];
        decision[i] = true;
    }
}

const houses = [];
for (let i = n - 1; i >= 0; ) {
    if (decision[i]) {
        houses.push(i);
        i -= 2;
    } else {
        i--;
    }
}

return { maxAmount: dp[n - 1], houses: houses.reverse() };
}

// Test
console.log(robWithHouses([2, 7, 9, 3, 1])); // Output: { maxAmount: 12,
houses: [0, 2, 4] }
console.log(robWithHouses([2, 1, 1, 2])); // Output: { maxAmount: 4, houses:
[0, 3] }
console.log(robWithHouses([1, 2, 3, 1])); // Output: { maxAmount: 4, houses:
[1, 3] }

```

### Explanation:

1. Use a `decision` array to track whether a house is robbed.
2. After calculating the maximum profit, backtrack through the `decision` array to identify the indices of the robbed houses.

### Time Complexity:

- **O(n)**: Single pass to calculate the DP array and another pass to backtrack.

### Space Complexity:

- **O(n)**: Space for the `dp` and `decision` arrays.

## House Robber - Circular Street (House Robber II)

**414. Problem:** The houses are arranged in a circular street. The first and last houses are adjacent, so you cannot rob both. Return the maximum amount that can be robbed.

### Solution:

```

function robCircular(nums) {
    const n = nums.length;
    if (n === 0) return 0;
    if (n === 1) return nums[0];

    function robLinear(start, end) {
        let prev2 = 0,
            prev1 = 0;
        for (let i = start; i <= end; i++) {
            const current = Math.max(prev1, prev2 + nums[i]);
            prev2 = prev1;
            prev1 = current;
        }
        return prev1;
    }

    return Math.max(robLinear(0, n - 2), robLinear(1, n - 1));
}

// Test
console.log(robCircular([2, 3, 2])); // Output: 3
console.log(robCircular([1, 2, 3, 1])); // Output: 4
console.log(robCircular([1, 2, 3])); // Output: 3

```

### **Explanation:**

1. Treat the problem as two linear house robber problems.
2. Compute the maximum profit for the two cases and return the larger value.

### **Time Complexity:**

- **O(n):** Each linear problem is solved in  $O(n)$ .

### **Space Complexity:**

- **O(1):** Space for two variables.

## **House Robber - Houses with Specific Gaps**

**415. Problem:** You are given a list of houses and a fixed gap  $k$  such that you can rob two houses only if there is a gap of at least  $k$  houses between them. Return the maximum amount that can be robbed.

### **Solution:**

```

function robWithGap(nums, k) {
    const n = nums.length;
    if (n === 0) return 0;

```

```

const dp = new Array(n).fill(0);
dp[0] = nums[0];

for (let i = 1; i < n; i++) {
    const robCurrent = nums[i] + (i > k ? dp[i - k - 1] : 0);
    dp[i] = Math.max(dp[i - 1], robCurrent);
}

return dp[n - 1];
}

// Test
console.log(robWithGap([2, 7, 9, 3, 1], 1)); // Output: 12
console.log(robWithGap([2, 1, 1, 2], 2)); // Output: 3
console.log(robWithGap([1, 2, 3, 1], 3)); // Output: 3

```

### **Explanation:**

1. Update the DP array based on whether the current house is robbed or skipped.
2. Incorporate the gap condition when updating the DP table.

### **Time Complexity:**

- **O(n):** Iterate through the houses once.

### **Space Complexity:**

- **O(n):** Space for the DP array.

## **House Robber - Houses with Variable Gaps**

### **416. Problem:**

Each house has a different gap requirement. You are given an array `gaps` where `gaps[i]` is the minimum number of houses you must skip after robbing house `i`. Return the maximum amount that can be robbed.

### **Solution:**

```

function robWithVariableGaps(nums, gaps) {
    const n = nums.length;
    if (n === 0) return 0;

    const dp = new Array(n).fill(0);
    dp[0] = nums[0];

    for (let i = 1; i < n; i++) {
        const robCurrent = nums[i] + (i > gaps[i] ? dp[i - gaps[i] - 1] : 0);
        dp[i] = Math.max(dp[i - 1], robCurrent);
    }

    return dp[n - 1];
}

```

```

        dp[i] = Math.max(dp[i - 1], robCurrent);
    }

    return dp[n - 1];
}

// Test
console.log(robWithVariableGaps([2, 7, 9, 3, 1], [1, 1, 2, 2, 1])); // Output:
12
console.log(robWithVariableGaps([2, 1, 1, 2], [0, 0, 1, 2])); // Output: 4
console.log(robWithVariableGaps([1, 2, 3, 1], [1, 1, 1, 1])); // Output: 4

```

### Explanation:

1. Use the `gaps` array to determine the houses that must be skipped after robbing a particular house.
2. Update the DP array accordingly.

### Time Complexity:

- **O(n)**: Iterate through the houses once.

### Space Complexity:

- **O(n)**: Space for the DP array.

## House Robber - Robbing Houses in a Binary Tree

### 417. Problem:

The houses are arranged in the form of a binary tree. A robber cannot rob two directly connected houses (parent-child relationship). Return the maximum amount that can be robbed.

### Solution:

```

function robBinaryTree(root) {
    function helper(node) {
        if (!node) return [0, 0];

        const left = helper(node.left);
        const right = helper(node.right);

        const robCurrent = node.val + left[1] + right[1];
        const skipCurrent =
            Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

        return [robCurrent, skipCurrent];
    }
}

```

```

const result = helper(root);
return Math.max(result[0], result[1]);
}

// TreeNode definition
class TreeNode {
  constructor(val, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

// Test
const root = new TreeNode(
  3,
  new TreeNode(2, null, new TreeNode(3)),
  new TreeNode(3, null, new TreeNode(1))
);
console.log(robBinaryTree(root)); // Output: 7

```

### Explanation:

1. Use a helper function that returns two values:
  - o Maximum profit if the current node is robbed.
  - o Maximum profit if the current node is skipped.
2. Recursively calculate values for child nodes and aggregate results.

### Time Complexity:

- **O(n):** Each node is visited once.

### Space Complexity:

- **O(h):** Space for the recursion stack, where  $h$  is the height of the tree.

## House Robber - Circular Street with Variable Gaps

**418. Problem:** The houses are arranged in a circular street, and each house has a variable gap requirement. Return the maximum amount that can be robbed.

### Solution:

```

function robCircularWithGaps(nums, gaps) {
  const n = nums.length;
  if (n === 0) return 0;
  if (n === 1) return nums[0];
}

```

```

function robLinearWithGaps(start, end) {
    const dp = new Array(end - start + 2).fill(0);
    for (let i = start; i <= end; i++) {
        const gap = gaps[i] || 0;
        const robCurrent =
            nums[i] + (i > start + gap ? dp[i - start - gap - 1] : 0);
        dp[i - start + 1] = Math.max(dp[i - start], robCurrent);
    }
    return dp[dp.length - 1];
}

return Math.max(robLinearWithGaps(0, n - 2), robLinearWithGaps(1, n - 1));
}

// Test
console.log(robCircularWithGaps([2, 3, 2], [1, 1, 1])); // Output: 3
console.log(robCircularWithGaps([1, 2, 3, 1], [1, 1, 1, 1])); // Output: 4
console.log(robCircularWithGaps([1, 2, 3], [1, 1, 1])); // Output: 3

```

### Explanation:

1. Treat the problem as two linear house robber problems with variable gaps.
2. Compute the maximum profit for each case and return the larger value.

### Time Complexity:

- **O(n):** Each linear problem is solved in  $O(n)$ .

### Space Complexity:

- **O(n):** Space for the DP array.

## House Robber - Robbing Alternating Houses

**419. Problem:** **Rob houses such that no two consecutively indexed houses are robbed, and the indices of robbed houses must alternate between even and odd indices. Return the maximum amount that can be robbed.**

### Solution:

```

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

    let evenSum = 0,
        oddSum = 0;

    for (let i = 0; i < nums.length; i++) {

```

```

        if (i % 2 === 0) {
            evenSum += nums[i];
        } else {
            oddSum += nums[i];
        }
    }

    return Math.max(evenSum, oddSum);
}

// Test
console.log(robAlternatingHouses([2, 7, 9, 3, 1])); // Output: 12 (even
indices: 2 + 9 + 1)
console.log(robAlternatingHouses([2, 1, 1, 2])); // Output: 3 (odd indices: 1
+ 2)
console.log(robAlternatingHouses([1, 2, 3, 1])); // Output: 4 (even indices: 1
+ 3)

```

### Explanation:

1. Calculate the sum of houses at even indices and the sum of houses at odd indices.
2. Return the larger sum.

### Time Complexity:

- **O(n):** Iterate through the array once.

### Space Complexity:

- **O(1):** Only two variables are used.

## House Robber - Skipping Houses with Custom Conditions

### 420. Problem:

**Rob houses where each house has a custom condition: the maximum number of adjacent houses that can be skipped is provided in an array `skips`. Return the maximum amount that can be robbed.**

### Solution:

```

function robWithCustomSkips(nums, skips) {
    const n = nums.length;
    if (n === 0) return 0;

    const dp = new Array(n).fill(0);
    dp[0] = nums[0];

    for (let i = 1; i < n; i++) {

```

```

        const skip = skips[i] || 0;
        const robCurrent = nums[i] + (i > skip ? dp[i - skip - 1] : 0);
        dp[i] = Math.max(dp[i - 1], robCurrent);
    }

    return dp[n - 1];
}

// Test
console.log(robWithCustomSkips([2, 7, 9, 3, 1], [1, 1, 2, 2, 1])); // Output: 12
console.log(robWithCustomSkips([2, 1, 1, 2], [0, 0, 1, 2])); // Output: 4
console.log(robWithCustomSkips([1, 2, 3, 1], [1, 1, 1, 1])); // Output: 4

```

### Explanation:

1. Use the `skips` array to determine the maximum adjacent houses that can be skipped after robbing a house.
2. Update the DP array based on whether the current house is robbed or skipped.

### Time Complexity:

- **O(n)**: Iterate through the houses once.

### Space Complexity:

- **O(n)**: Space for the DP array.

## Decode Ways - Count Total Decodings

### 421. Problem:

A message containing letters from A-Z is encoded using the following mapping:

- 'A' -> 1, 'B' -> 2, ..., 'Z' -> 26.

Given a string s containing only digits, return the total number of ways to decode it.

```

Input: s = "12"
Output: 2
Explanation: "12" can be decoded as "AB" (1 2) or "L" (12).

Input: s = "226"
Output: 3
Explanation: "226" can be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

```

### Solution:

```
function numDecodings(s) {
    if (s[0] === "0") return 0;

    const n = s.length;
    const dp = new Array(n + 1).fill(0);
    dp[0] = 1;
    dp[1] = s[0] !== "0" ? 1 : 0;

    for (let i = 2; i <= n; i++) {
        if (s[i - 1] !== "0") {
            dp[i] += dp[i - 1];
        }
        const twoDigit = parseInt(s.substring(i - 2, i));
        if (twoDigit >= 10 && twoDigit <= 26) {
            dp[i] += dp[i - 2];
        }
    }

    return dp[n];
}

// Test
console.log(numDecodings("12")); // Output: 2
console.log(numDecodings("226")); // Output: 3
console.log(numDecodings("0")); // Output: 0
console.log(numDecodings("06")); // Output: 0
```

### Explanation:

1. Use a DP array to calculate the number of decodings up to each index.
2. At each step:
  - o Include the current digit if it's valid (1-9).
  - o Include the last two digits if they form a valid number (10-26).

### Time Complexity:

- **O(n):** Single pass through the string.

### Space Complexity:

- **O(n):** Space for the DP array.

## Decode Ways - Space Optimized

### 422. Problem:

**Optimize the space complexity of the solution to O(1) by eliminating the DP array.**

### Solution:

```
function numDecodingsOptimized(s) {  
    if (s[0] === "0") return 0;  
  
    let prev2 = 1,  
        prev1 = 1; // dp[i-2] and dp[i-1]  
  
    for (let i = 1; i < s.length; i++) {  
        let current = 0;  
        if (s[i] !== "0") {  
            current += prev1;  
        }  
        const twoDigit = parseInt(s.substring(i - 1, i + 1));  
        if (twoDigit >= 10 && twoDigit <= 26) {  
            current += prev2;  
        }  
        prev2 = prev1;  
        prev1 = current;  
    }  
  
    return prev1;  
}  
  
// Test  
console.log(numDecodingsOptimized("12")); // Output: 2  
console.log(numDecodingsOptimized("226")); // Output: 3  
console.log(numDecodingsOptimized("0")); // Output: 0  
console.log(numDecodingsOptimized("06")); // Output: 0
```

### Explanation:

1. Use two variables to store the results of the last two computations.
2. Update these variables iteratively without using additional space.

### Time Complexity:

- **O(n):** Single pass through the string.

### Space Complexity:

- **O(1):** Only two variables are used.

## Decode Ways - Return All Decodings

**423. Problem: Return all possible decodings of the string  $s$  instead of just the count.**

### Solution:

```

function allDecodings(s) {
  if (s[0] === "0") return [];

  const result = [];
  const map = Array.from({ length: 26 }, (_, i) => String.fromCharCode(65 + i));

  function backtrack(index, path) {
    if (index === s.length) {
      result.push(path.join(""));
      return;
    }

    // Single digit
    const oneDigit = parseInt(s[index]);
    if (oneDigit >= 1 && oneDigit <= 9) {
      path.push(map[oneDigit - 1]);
      backtrack(index + 1, path);
      path.pop();
    }

    // Two digits
    if (index + 1 < s.length) {
      const twoDigit = parseInt(s.substring(index, index + 2));
      if (twoDigit >= 10 && twoDigit <= 26) {
        path.push(map[twoDigit - 1]);
        backtrack(index + 2, path);
        path.pop();
      }
    }
  }

  backtrack(0, []);
  return result;
}

// Test
console.log(allDecodings("12")); // Output: ["AB", "L"]
console.log(allDecodings("226")); // Output: ["BZ", "VF", "BBF"]
console.log(allDecodings("0")); // Output: []
console.log(allDecodings("06")); // Output: []

```

### **Explanation:**

1. Use a helper function to explore all possible decodings recursively.
2. At each step, append valid characters to the current path and backtrack.

### **Time Complexity:**

- **O(2^n)**: Exponential due to backtracking.

### Space Complexity:

- **O(n)**: Space for the recursion stack.

## Decode Ways - Decode with '\*' Wildcard

### 424. Problem:

In addition to digits 1-9, the string s may contain the '\*' character, which can represent any digit from 1-9. Return the total number of ways to decode the string s. The result should be modulo  $10^9 + 7$ .

### Rules for '\*':

- '\*' can be any single digit (1-9).
- '\*' followed by another digit can represent any two-digit number (10-26).

### Solution:

```
function numDecodingsWildcard(s) {
    const MOD = 1e9 + 7;
    const n = s.length;
    if (s[0] === "0") return 0;

    let prev2 = 1,
        prev1 = s[0] === "*" ? 9 : 1;

    for (let i = 1; i < n; i++) {
        let current = 0;

        if (s[i] === "*") {
            current += 9 * prev1; // '*' as single digit
        } else if (s[i] !== "0") {
            current += prev1; // Regular single digit
        }

        if (s[i - 1] === "*") {
            if (s[i] === "*") {
                current += 15 * prev2; // '**' can be 11-19 or 21-26
            } else if (s[i] <= "6") {
                current += 2 * prev2; // '*X' where X <= 6
            } else {
                current += prev2; // '*X' where X > 6
            }
        } else if (s[i - 1] === "1") {
            current += s[i] === "*" ? 9 * prev2 : prev2; // '1*' or '1X'
        } else if (s[i - 1] === "2") {
```

```

        current += s[i] === "*" ? 6 * prev2 : s[i] <= "6" ? prev2 : 0; // '2*' or '2X'
    }

    current %= MOD;
    prev2 = prev1;
    prev1 = current;
}

return prev1;
}

// Test
console.log(numDecodingsWildcard("1*")); // Output: 18
console.log(numDecodingsWildcard("**")); // Output: 96
console.log(numDecodingsWildcard("2*")); // Output: 15
console.log(numDecodingsWildcard("**1")); // Output: 180

```

### Explanation:

1. Extend the rules for '\*' to account for all possible valid decodings.
2. Use two variables to optimize space and compute results iteratively.

### Time Complexity:

- **O(n)**: Single pass through the string.

### Space Complexity:

- **O(1)**: Only two variables are used.

## Decode Ways - Count Decodings with Constraints

### 425. Problem:

Given a string `s`, return the number of ways to decode it such that no decoded letter appears more than once. If no valid decoding exists, return 0.

### Solution:

```

function numUniqueDecodings(s) {
  const result = new Set();
  const map = Array.from({ length: 26 }, (_, i) => String.fromCharCode(65 + i));

  function backtrack(index, path, used) {
    if (index === s.length) {
      result.add(path.join(""));
      return;
    }
  }
}

```

```

    }

    // Single digit
    const oneDigit = parseInt(s[index]);
    if (oneDigit >= 1 && oneDigit <= 9 && !used.has(map[oneDigit - 1])) {
        used.add(map[oneDigit - 1]);
        path.push(map[oneDigit - 1]);
        backtrack(index + 1, path, used);
        path.pop();
        used.delete(map[oneDigit - 1]);
    }

    // Two digits
    if (index + 1 < s.length) {
        const twoDigit = parseInt(s.substring(index, index + 2));
        if (twoDigit >= 10 && twoDigit <= 26 && !used.has(map[twoDigit - 1])) {
            used.add(map[twoDigit - 1]);
            path.push(map[twoDigit - 1]);
            backtrack(index + 2, path, used);
            path.pop();
            used.delete(map[twoDigit - 1]);
        }
    }
}

backtrack(0, [], new Set());
return result.size;
}

// Test
console.log(numUniqueDecodings("12")); // Output: 2 ("AB", "L")
console.log(numUniqueDecodings("226")); // Output: 3 ("BZ", "VF", "BBF")
console.log(numUniqueDecodings("12345")); // Output: 6

```

### Explanation:

1. Use a set to track already used characters in the current decoding path.
2. Backtrack to explore all valid decoding paths while ensuring no duplicates.

### Time Complexity:

- **O( $2^n$ ):** Exponential due to backtracking.

### Space Complexity:

- **O(n):** Space for the recursion stack.

## Decode Ways - Minimum Decoding Length

#### 426. Problem:

Find the minimum number of characters required to decode the entire string  $s$ . Each decoding step must decode one or two characters.

#### Solution:

```
function minDecodingLength(s) {
    const n = s.length;
    if (s[0] === "0") return -1;

    const dp = new Array(n + 1).fill(Infinity);
    dp[0] = 0;
    dp[1] = s[0] !== "0" ? 1 : Infinity;

    for (let i = 2; i <= n; i++) {
        if (s[i - 1] !== "0") {
            dp[i] = Math.min(dp[i], dp[i - 1] + 1);
        }
        const twoDigit = parseInt(s.substring(i - 2, i));
        if (twoDigit >= 10 && twoDigit <= 26) {
            dp[i] = Math.min(dp[i], dp[i - 2] + 1);
        }
    }

    return dp[n] === Infinity ? -1 : dp[n];
}

// Test
console.log(minDecodingLength("12")); // Output: 1
console.log(minDecodingLength("226")); // Output: 2
console.log(minDecodingLength("0")); // Output: -1
console.log(minDecodingLength("06")); // Output: -1
```

#### Explanation:

1. Use a DP array to track the minimum decoding length for each prefix of the string.
2. At each step:
  - o Add 1 for single-character decodings.
  - o Add 1 for valid two-character decodings.

#### Time Complexity:

- **O(n):** Single pass through the string.

#### Space Complexity:

- **O(n):** Space for the DP array.

## Decode Ways - Maximum Decoding Length

**427. Problem:** Find the maximum number of decoding steps required to decode the string  $s$ . Each decoding step must decode one or two characters.

**Solution:**

```
function maxDecodingLength(s) {
    const n = s.length;
    if (s[0] === "0") return 0;

    const dp = new Array(n + 1).fill(0);
    dp[0] = 0;
    dp[1] = s[0] !== "0" ? 1 : 0;

    for (let i = 2; i <= n; i++) {
        if (s[i - 1] !== "0") {
            dp[i] = Math.max(dp[i], dp[i - 1] + 1);
        }
        const twoDigit = parseInt(s.substring(i - 2, i));
        if (twoDigit >= 10 && twoDigit <= 26) {
            dp[i] = Math.max(dp[i], dp[i - 2] + 1);
        }
    }

    return dp[n];
}

// Test
console.log(maxDecodingLength("12")); // Output: 2
console.log(maxDecodingLength("226")); // Output: 3
console.log(maxDecodingLength("0")); // Output: 0
console.log(maxDecodingLength("06")); // Output: 0
```

**Explanation:**

1. Use a DP array to calculate the maximum decoding steps for each prefix.
2. At each step, maximize the decoding steps for single-character and two-character decodings.

**Time Complexity:**

- **O(n):** Single pass through the string.

**Space Complexity:**

- **O(n):** Space for the DP array.

## Decode Ways - Decode with a Cost

### 428. Problem:

Each decoding step has a cost. Single-character decodings cost 1, and two-character decodings cost 2. Find the minimum cost to decode the string s.

### Solution:

```
function minDecodingCost(s) {
    const n = s.length;
    if (s[0] === "0") return -1;

    const dp = new Array(n + 1).fill(Infinity);
    dp[0] = 0;
    dp[1] = s[0] !== "0" ? 1 : Infinity;

    for (let i = 2; i <= n; i++) {
        if (s[i - 1] !== "0") {
            dp[i] = Math.min(dp[i], dp[i - 1] + 1);
        }
        const twoDigit = parseInt(s.substring(i - 2, i));
        if (twoDigit >= 10 && twoDigit <= 26) {
            dp[i] = Math.min(dp[i], dp[i - 2] + 2);
        }
    }

    return dp[n] === Infinity ? -1 : dp[n];
}

// Test
console.log(minDecodingCost("12")); // Output: 1
console.log(minDecodingCost("226")); // Output: 3
console.log(minDecodingCost("0")); // Output: -1
console.log(minDecodingCost("06")); // Output: -1
```

### Explanation:

1. Compute the cost for single-character and two-character decodings at each step.
2. Use a DP array to store the minimum cost.

### Time Complexity:

- **O(n):** Single pass through the string.

### Space Complexity:

- **O(n):** Space for the DP array.

## Decode Ways - Decode with Maximum Cost

**429. Problem:** Each decoding step has a cost. Single-character decodings cost 1, and two-character decodings cost 2. Find the maximum cost to decode the string s.

**Solution:**

```
function maxDecodingCost(s) {
    const n = s.length;
    if (s[0] === "0") return 0;

    const dp = new Array(n + 1).fill(0);
    dp[0] = 0;
    dp[1] = s[0] !== "0" ? 1 : 0;

    for (let i = 2; i <= n; i++) {
        if (s[i - 1] !== "0") {
            dp[i] = Math.max(dp[i], dp[i - 1] + 1);
        }
        const twoDigit = parseInt(s.substring(i - 2, i));
        if (twoDigit >= 10 && twoDigit <= 26) {
            dp[i] = Math.max(dp[i], dp[i - 2] + 2);
        }
    }

    return dp[n];
}

// Test
console.log(maxDecodingCost("12")); // Output: 3
console.log(maxDecodingCost("226")); // Output: 5
console.log(maxDecodingCost("0")); // Output: 0
console.log(maxDecodingCost("06")); // Output: 0
```

**Explanation:**

1. Use a DP array to compute the maximum cost to decode each prefix of the string.
2. At each step, consider the cost of single-character and two-character decodings.

**Time Complexity:**

- **O(n):** Iterate through the string once.

**Space Complexity:**

- **O(n):** Space for the DP array.

## Decode Ways - Decode with Restricted Two-Digit Decodings

**430. Problem:** Decode the string  $s$  such that only specific two-digit numbers are allowed for two-character decodings. Given an array `allowed`, return the total number of valid decodings.

**Solution:**

```
function decodeWithRestrictedPairs(s, allowed) {
    const n = s.length;
    if (s[0] === "0") return 0;

    const allowedSet = new Set(allowed.map(String));
    const dp = new Array(n + 1).fill(0);
    dp[0] = 1;
    dp[1] = s[0] !== "0" ? 1 : 0;

    for (let i = 2; i <= n; i++) {
        if (s[i - 1] !== "0") {
            dp[i] += dp[i - 1];
        }
        const twoDigit = s.substring(i - 2, i);
        if (allowedSet.has(twoDigit)) {
            dp[i] += dp[i - 2];
        }
    }

    return dp[n];
}

// Test
console.log(decodeWithRestrictedPairs("12", [10, 12, 21])); // Output: 2
("AB", "L")
console.log(decodeWithRestrictedPairs("226", [10, 20, 26])); // Output: 1
("BZ")
console.log(decodeWithRestrictedPairs("123", [12, 23])); // Output: 2 ("LC",
"AW")
console.log(decodeWithRestrictedPairs("0", [10, 20])); // Output: 0
```

**Explanation:**

1. Use a `Set` for efficient lookups of valid two-digit decodings.
2. Update the DP array based on single-character decodings and allowed two-character decodings.

**Time Complexity:**

- **O(n):** Single pass through the string.

## Space Complexity:

- **O(n)**: Space for the DP array.

## Unique Paths - Grid with No Obstacles

### 431. Problem:

You are given an  $m \times n$  grid. You are a robot located at the top-left corner of the grid. Your goal is to reach the bottom-right corner of the grid. The robot can only move either down or right at any point in time.

Return the number of unique paths the robot can take to reach the bottom-right corner.

```
Input: m = 3, n = 7
Output: 28
```

```
Input: m = 3, n = 2
```

```
Output: 3
```

Explanation:

From the top-left corner, there are three possible paths to the bottom-right corner:

1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down

## Solution:

```
function uniquePaths(m, n) {
    const dp = Array.from({ length: m }, () => Array(n).fill(0));

    // Initialize first row and first column
    for (let i = 0; i < m; i++) dp[i][0] = 1;
    for (let j = 0; j < n; j++) dp[0][j] = 1;

    // Fill the DP table
    for (let i = 1; i < m; i++) {
        for (let j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }

    return dp[m - 1][n - 1];
}

// Test
console.log(uniquePaths(3, 7)); // Output: 28
console.log(uniquePaths(3, 2)); // Output: 3
```

```
console.log(uniquePaths(7, 3)); // Output: 28
```

### Explanation:

1. The robot can reach any cell  $(i, j)$  by either:
  - o Moving down from the cell above  $(i-1, j)$ .
  - o Moving right from the cell to the left  $(i, j-1)$ .
2. Base case: The first row and column can only be reached in one way.
3. Compute the number of paths for each cell iteratively using the DP formula.

### Time Complexity:

- $O(m * n)$ : Fill the entire DP table.

### Space Complexity:

- $O(m * n)$ : Space for the DP table.

## Unique Paths - Space Optimized

432. Problem: Optimize the space complexity of the solution to  $O(n)$ .

### Solution:

```
function uniquePathsOptimized(m, n) {  
    const dp = Array(n).fill(1);  
  
    for (let i = 1; i < m; i++) {  
        for (let j = 1; j < n; j++) {  
            dp[j] += dp[j - 1];  
        }  
    }  
  
    return dp[n - 1];  
}  
  
// Test  
console.log(uniquePathsOptimized(3, 7)); // Output: 28  
console.log(uniquePathsOptimized(3, 2)); // Output: 3  
console.log(uniquePathsOptimized(7, 3)); // Output: 28
```

### Explanation:

1. Use a single row to compute the unique paths for each column iteratively.
2. Update the current cell by adding the value of the cell to its left.

### Time Complexity:

- $O(m * n)$ : Iterate through the grid.

### Space Complexity:

- $O(n)$ : Space for the DP array.

## Unique Paths - Grid with Obstacles

### 433. Problem:

Now consider some obstacles in the  $m \times n$  grid. An obstacle and space are marked as 1 and 0 respectively in a 2D array `obstacleGrid`. Return the number of unique paths that the robot can take to reach the bottom-right corner.

The robot cannot move through obstacles.

```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation:
There is one obstacle in the middle of the grid.
From the top-left corner, there are two paths to the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

Input: obstacleGrid = [[0,1],[0,0]]
Output: 1
```

### Solution:

```
function uniquePathsWithObstacles(obstacleGrid) {
    const m = obstacleGrid.length,
        n = obstacleGrid[0].length;
    const dp = Array.from({ length: m }, () => Array(n).fill(0));

    // Initialize first row
    for (let i = 0; i < m; i++) {
        if (obstacleGrid[i][0] === 1) break;
        dp[i][0] = 1;
    }

    // Initialize first column
    for (let j = 0; j < n; j++) {
        if (obstacleGrid[0][j] === 1) break;
        dp[0][j] = 1;
    }
}
```

```

// Fill the DP table
for (let i = 1; i < m; i++) {
  for (let j = 1; j < n; j++) {
    if (obstacleGrid[i][j] === 1) {
      dp[i][j] = 0;
    } else {
      dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
    }
  }
}

return dp[m - 1][n - 1];
}

// Test
console.log(
  uniquePathsWithObstacles([
    [0, 0, 0],
    [0, 1, 0],
    [0, 0, 0],
  ])
); // Output: 2
console.log(
  uniquePathsWithObstacles([
    [0, 1],
    [0, 0],
  ])
); // Output: 1

```

### **Explanation:**

1. If the robot encounters an obstacle, it cannot move through that cell.
2. Set the paths for cells with obstacles to 0 and calculate paths for the rest of the grid using the DP formula.

### **Time Complexity:**

- **O(m \* n):** Iterate through the grid.

### **Space Complexity:**

- **O(m \* n):** Space for the DP table.

## **Unique Paths - Space Optimized with Obstacles**

### **434. Problem:**

**Optimize the space complexity of the grid with obstacles to O(n).**

### Solution:

```
function uniquePathsWithObstaclesOptimized(obstacleGrid) {
    const m = obstacleGrid.length,
        n = obstacleGrid[0].length;
    const dp = Array(n).fill(0);
    dp[0] = obstacleGrid[0][0] === 0 ? 1 : 0;

    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (obstacleGrid[i][j] === 1) {
                dp[j] = 0;
            } else if (j > 0) {
                dp[j] += dp[j - 1];
            }
        }
    }

    return dp[n - 1];
}

// Test
console.log(
    uniquePathsWithObstaclesOptimized([
        [0, 0, 0],
        [0, 1, 0],
        [0, 0, 0],
    ])
); // Output: 2
console.log(
    uniquePathsWithObstaclesOptimized([
        [0, 1],
        [0, 0],
    ])
); // Output: 1
```

### Explanation:

1. Use a single row to calculate the number of unique paths iteratively.
2. Reset the paths to 0 for cells with obstacles.

### Time Complexity:

- **O(m \* n):** Iterate through the grid.

### Space Complexity:

- **O(n):** Space for the DP array.

## Unique Paths - Count All Paths (Backtracking)

435. Problem: Count all unique paths from the top-left to the bottom-right corner using backtracking instead of DP.

Solution:

```
function uniquePathsBacktracking(m, n) {  
    function backtrack(i, j) {  
        if (i === m - 1 && j === n - 1) return 1;  
        if (i >= m || j >= n) return 0;  
        return backtrack(i + 1, j) + backtrack(i, j + 1);  
    }  
    return backtrack(0, 0);  
  
// Test  
console.log(uniquePathsBacktracking(3, 7)); // Output: 28  
console.log(uniquePathsBacktracking(3, 2)); // Output: 3
```

Explanation:

1. Use recursion to explore all valid paths.
2. At each step, the robot can move either right or down until it reaches the target.

Time Complexity:

- $O(2^{m+n})$ : Exponential due to recursive exploration.

Space Complexity:

- $O(m + n)$ : Space for the recursion stack.

## Unique Paths - Count Paths Using Combinatorics

436. Problem: Count the number of unique paths from the top-left to the bottom-right corner of an  $m \times n$  grid using combinatorics.

Solution:

```
function uniquePathsCombinatorics(m, n) {  
    function factorial(num) {  
        let result = 1;  
        for (let i = 1; i <= num; i++) {  
            result *= i;  
        }  
        return result;  
    }
```

```

const totalMoves = m + n - 2;
const downMoves = m - 1;
return (
  factorial(totalMoves) /
  (factorial(downMoves) * factorial(totalMoves - downMoves))
);
}

// Test
console.log(uniquePathsCombinatorics(3, 7)); // Output: 28
console.log(uniquePathsCombinatorics(3, 2)); // Output: 3
console.log(uniquePathsCombinatorics(7, 3)); // Output: 28

```

### Explanation:

1. To reach the bottom-right corner, the robot needs to take exactly  $(m-1)$  down moves and  $(n-1)$  right moves, in any order.
2. The total number of unique paths is the number of ways to arrange these moves:
  - o Total moves =  $(m-1) + (n-1) = m + n - 2$ .
  - o Choose  $(m-1)$  down moves from the total:  

$$\text{Unique Paths} = \frac{(m+n-2)!}{(m-1)!(n-1)!}$$

$$= \frac{\binom{m+n-2}{m-1}}{\frac{(m+n-2)!}{(m-1)!(n-1)!}}$$

$$= \frac{(m-1)!(n-1)!(m+n-2)!}{(m-1)!(n-1)!(m+n-2)!}$$

$$= 1$$

### Time Complexity:

- $O(m + n)$ : Factorial computation.

### Space Complexity:

- $O(1)$ : No additional memory used.

## Unique Paths - Grid with Fixed Starting and Ending Points

**437. Problem:** Count the number of unique paths from a given start point  $(sx, sy)$  to a given end point  $(ex, ey)$  in an  $m \times n$  grid.

### Solution:

```

function uniquePathsSubgrid(m, n, sx, sy, ex, ey) {
  const rows = ex - sx + 1;
  const cols = ey - sy + 1;

  function factorial(num) {
    let result = 1;
    for (let i = 1; i <= num; i++) {
      result *= i;
    }
  }
}

```

```

        return result;
    }

    const totalMoves = rows + cols - 2;
    const downMoves = rows - 1;
    return (
        factorial(totalMoves) /
        (factorial(downMoves) * factorial(totalMoves - downMoves))
    );
}

// Test
console.log(uniquePathsSubgrid(7, 7, 0, 0, 2, 3)); // Output: 10
console.log(uniquePathsSubgrid(7, 7, 1, 1, 5, 5)); // Output: 70

```

### Explanation:

1. Treat the specified subgrid as a new grid with dimensions  $(ex - sx + 1) \times (ey - sy + 1)$ .
2. Use combinatorics to calculate the number of paths for this subgrid.

### Time Complexity:

- $O(m + n)$ : Factorial computation.

### Space Complexity:

- $O(1)$ : No additional memory used.

## Unique Paths - Multiple Obstacles and Detours

**438. Problem:** Count the number of unique paths from the top-left to the bottom-right corner, but the robot must visit specific intermediate cells before reaching the target.

### Solution:

```

function uniquePathsWithIntermediatePoints(m, n, intermediatePoints) {
    function combinatorics(m, n) {
        function factorial(num) {
            let result = 1;
            for (let i = 1; i <= num; i++) {
                result *= i;
            }
            return result;
        }

        const totalMoves = m + n - 2;
        const downMoves = m - 1;
    }
}

```

```

        return (
            factorial(totalMoves) /
            (factorial(downMoves) * factorial(totalMoves - downMoves))
        );
    }

    let totalPaths = 1;
    let points = [[0, 0], ...intermediatePoints, [m - 1, n - 1]];

    for (let i = 0; i < points.length - 1; i++) {
        const [sx, sy] = points[i];
        const [ex, ey] = points[i + 1];
        const rows = ex - sx + 1;
        const cols = ey - sy + 1;
        totalPaths *= combinatorics(rows, cols);
    }

    return totalPaths;
}

// Test
console.log(uniquePathsWithIntermediatePoints(5, 5, [[2, 2]])); // Output: 6
console.log(
    uniquePathsWithIntermediatePoints(5, 5, [
        [1, 1],
        [3, 3],
    ])
); // Output: 36

```

### Explanation:

1. Divide the grid into segments defined by the intermediate points.
2. Calculate the paths for each segment using combinatorics and multiply the results.

### Time Complexity:

- **O(k \* (m + n)):** Where k is the number of intermediate points.

### Space Complexity:

- **O(1):** No additional memory used.

## Unique Paths - Count Paths with Maximum Turns

**439. Problem:** Count the number of unique paths from the top-left to the bottom-right corner of an  $m \times n$  grid, but with a constraint on the maximum number of turns the robot can make. A turn occurs when the robot switches its direction from "right" to "down" or "down" to "right."

### Solution:

```
function uniquePathsWithMaxTurns(m, n, maxTurns) {
  const dp = Array.from({ length: m }, () =>
    Array.from({ length: n }, () => Array(maxTurns + 1).fill(0))
  );

  // Base cases
  for (let i = 0; i < m; i++) dp[i][0][0] = 1;
  for (let j = 0; j < n; j++) dp[0][j][0] = 1;

  // Fill the DP table
  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      for (let k = 0; k <= maxTurns; k++) {
        dp[i][j][k] += dp[i - 1][j][k];
        if (k > 0) dp[i][j][k] += dp[i][j - 1][k - 1];
      }
    }
  }

  // Sum all paths with turns <= maxTurns
  let result = 0;
  for (let k = 0; k <= maxTurns; k++) {
    result += dp[m - 1][n - 1][k];
  }

  return result;
}

// Test
console.log(uniquePathsWithMaxTurns(3, 3, 2)); // Output: 6
console.log(uniquePathsWithMaxTurns(4, 4, 3)); // Output: 20
```

### Explanation:

1. Use a 3D DP table to track paths for each cell with a limited number of turns.
2. At each step, decide whether to continue in the same direction or make a turn.
3. Sum all paths with turns less than or equal to the given maximum.

### Time Complexity:

- **O(m \* n \* maxTurns):** Iterate through all cells and turn states.

### Space Complexity:

- **O(m \* n \* maxTurns):** Space for the 3D DP table.

## Unique Paths - Count Paths with Minimum Turns

**440. Problem:** Count the number of unique paths from the top-left to the bottom-right corner of an  $m \times n$  grid that use the **minimum number of turns** possible.

**Solution:**

```
function uniquePathsMinTurns(m, n) {
    const minTurns = Math.min(m - 1, n - 1);

    function factorial(num) {
        let result = 1;
        for (let i = 1; i <= num; i++) result *= i;
        return result;
    }

    const totalMoves = m + n - 2;
    const downMoves = m - 1;
    return (
        factorial(totalMoves) /
        (factorial(downMoves) * factorial(totalMoves - downMoves))
    );
}

// Test
console.log(uniquePathsMinTurns(3, 3)); // Output: 6
console.log(uniquePathsMinTurns(4, 4)); // Output: 20
```

**Explanation:**

1. Minimize the number of turns by prioritizing one direction until forced to turn.
2. Use combinatorics to calculate the number of unique paths.

**Time Complexity:**

- **O( $m + n$ ):** Factorial computation.

**Space Complexity:**

- **O(1):** No additional memory used.

## Predicting Jump Patterns in Arrays - Can Reach the Last Index

**441. Problem:** You are given an integer array `nums` where `nums[i]` represents the maximum jump length you can make from index `i`. Return `true` if you can reach the last index, or `false` otherwise.

```

Input: nums = [2, 3, 1, 1, 4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Input: nums = [3, 2, 1, 0, 4]
Output: false
Explanation: You will always arrive at index 3, where your maximum jump length
is 0. Therefore, you cannot move further.

```

### Solution:

```

function canJump(nums) {
    let farthest = 0;

    for (let i = 0; i < nums.length; i++) {
        if (i > farthest) return false; // If current index is beyond the farthest
        // reachable index
        farthest = Math.max(farthest, i + nums[i]);
        if (farthest >= nums.length - 1) return true; // If we can reach or exceed
        // the last index
    }

    return false;
}

// Test
console.log(canJump([2, 3, 1, 1, 4])); // Output: true
console.log(canJump([3, 2, 1, 0, 4])); // Output: false
console.log(canJump([0])); // Output: true
console.log(canJump([1, 0, 1])); // Output: false

```

### Explanation:

1. The `farthest` variable tracks the maximum index we can reach as we iterate through the array.
2. If at any point `i > farthest`, it means the current index is unreachable.
3. Return `true` if `farthest` reaches or exceeds the last index.

### Time Complexity:

- **O(n)**: Iterate through the array once.

### Space Complexity:

- **O(1)**: No extra space used.

## Predicting Jump Patterns in Arrays - Minimum Jumps to Reach the End

**442. Problem:** Given an array `nums`, return the minimum number of jumps required to reach the last index. Assume that you can always reach the last index.

Input: `nums = [2, 3, 1, 1, 4]`

Output: 2

Explanation: The minimum number of jumps to reach the last index is 2.

Jump 1 step from index 0 to 1, then 3 steps to the last index.

Input: `nums = [2, 3, 0, 1, 4]`

Output: 2

### Solution:

```
function jump(nums) {  
    let jumps = 0,  
        currentEnd = 0,  
        farthest = 0;  
  
    for (let i = 0; i < nums.length - 1; i++) {  
        farthest = Math.max(farthest, i + nums[i]);  
        if (i === currentEnd) {  
            jumps++;  
            currentEnd = farthest;  
        }  
    }  
  
    return jumps;  
}  
  
// Test  
console.log(jump([2, 3, 1, 1, 4])); // Output: 2  
console.log(jump([2, 3, 0, 1, 4])); // Output: 2  
console.log(jump([1, 2, 3, 4, 5])); // Output: 3
```

### Explanation:

1. The `farthest` variable tracks the farthest index reachable during each iteration.
2. When the current index exceeds `currentEnd`, it means a new jump is required to proceed further.
3. Increment the jump count and update `currentEnd` to `farthest`.

### Time Complexity:

- **O(n):** Iterate through the array once.

## Space Complexity:

- **O(1)**: No extra space used.

## Predicting Jump Patterns in Arrays - Maximum Reachable Index

**443. Problem:** Given an array `nums`, return the maximum index you can reach starting from index 0. If the index exceeds the length of the array, return the length of the array as the maximum reachable index.

```
Input: nums = [2, 3, 1, 1, 4]
Output: 4
Explanation: Starting from index 0, the maximum reachable index is the last
index (4).

Input: nums = [3, 2, 1, 0, 4]
Output: 3
Explanation: Starting from index 0, the maximum reachable index is index 3.

Input: nums = [1, 0, 0, 0]
Output: 1
Explanation: The maximum reachable index is index 1 because the first step
reaches only the first index.
```

## Solution:

```
function maxReachableIndex(nums) {
    let farthest = 0;

    for (let i = 0; i < nums.length; i++) {
        if (i > farthest) break; // If the current index is unreachable
        farthest = Math.max(farthest, i + nums[i]);
    }

    return Math.min(farthest, nums.length - 1);
}

// Test
console.log(maxReachableIndex([2, 3, 1, 1, 4])); // Output: 4
console.log(maxReachableIndex([3, 2, 1, 0, 4])); // Output: 3
console.log(maxReachableIndex([1, 0, 0, 0])); // Output: 1
console.log(maxReachableIndex([0])); // Output: 0
```

## Explanation:

1. Start from index 0 and calculate the farthest index reachable using `farthest = max(farthest, i + nums[i])`.
2. Stop the iteration if the current index exceeds the farthest reachable index.
3. Return the minimum of `farthest` and `nums.length - 1`.

### Time Complexity:

- **O(n)**: Single pass through the array.

### Space Complexity:

- **O(1)**: No additional space used.

## Predicting Jump Patterns in Arrays - Can Reach a Specific Index

**444. Problem: Given an array `nums` and a target index `t`, return `true` if you can reach index `t` starting from index 0. Otherwise, return `false`.**

### Solution:

```
function canReachIndex(nums, t) {
  let farthest = 0;

  for (let i = 0; i < nums.length; i++) {
    if (i > farthest) return false; // If the current index is unreachable
    farthest = Math.max(farthest, i + nums[i]);
    if (farthest >= t) return true; // If the target index is reachable
  }

  return false;
}

// Test
console.log(canReachIndex([2, 3, 1, 1, 4], 4)); // Output: true
console.log(canReachIndex([3, 2, 1, 0, 4], 4)); // Output: false
console.log(canReachIndex([1, 0, 0, 0], 2)); // Output: false
console.log(canReachIndex([0], 0)); // Output: true
```

### Explanation:

1. Use the `farthest` variable to track the farthest index reachable from the start.
2. If `farthest` becomes greater than or equal to the target index `t`, return `true`.
3. If the loop ends without reaching `t`, return `false`.

### Time Complexity:

- **O(n)**: Single pass through the array.

## Space Complexity:

- **O(1)**: No extra space used.

## Predicting Jump Patterns in Arrays - Minimum Jumps to Reach Any Index

**445. Problem:** Given an array `nums`, return an array `result` where `result[i]` is the minimum number of jumps required to reach index `i` from index 0.

### Solution:

```
function minJumpsToEachIndex(nums) {  
    const n = nums.length;  
    const jumps = Array(n).fill(Infinity);  
    jumps[0] = 0;  
  
    const queue = [0];  
  
    while (queue.length > 0) {  
        const index = queue.shift();  
        for (let i = 1; i <= nums[index]; i++) {  
            const next = index + i;  
            if (next < n && jumps[next] === Infinity) {  
                jumps[next] = jumps[index] + 1;  
                queue.push(next);  
            }  
        }  
    }  
  
    return jumps;  
}  
  
// Test  
console.log(minJumpsToEachIndex([2, 3, 1, 1, 4])); // Output: [0, 1, 1, 2, 2]  
console.log(minJumpsToEachIndex([3, 2, 1, 0, 4])); // Output: [0, 1, 1, 1,  
Infinity]  
console.log(minJumpsToEachIndex([1, 1, 1, 1])); // Output: [0, 1, 2, 3]
```

### Explanation:

1. Start from index 0 and initialize the jumps array with `Infinity` to represent unreachable indices.
2. Use a BFS approach to process each index and update the minimum jumps for its neighbors.
3. Return the resulting jumps array.

## Time Complexity:

- **O(n^2):** In the worst case, BFS processes every possible jump.

### Space Complexity:

- **O(n):** Space for the queue and jumps array.

## Predicting Jump Patterns in Arrays - Maximum Jumps to Reach Any Index

**446. Problem:** Given an array `nums`, return an array `result` where `result[i]` is the **maximum number of jumps** required to reach index `i` from index 0, assuming all jumps are valid.

### Solution:

```
function maxJumpsToEachIndex(nums) {
  const n = nums.length;
  const jumps = Array(n).fill(-Infinity); // -Infinity indicates not reachable initially
  jumps[0] = 0;

  const queue = [0];

  while (queue.length > 0) {
    const index = queue.shift();
    for (let i = 1; i <= nums[index]; i++) {
      const next = index + i;
      if (next < n) {
        jumps[next] = Math.max(jumps[next], jumps[index] + 1);
        if (!queue.includes(next)) {
          queue.push(next);
        }
      }
    }
  }

  return jumps.map((j) => (j === -Infinity ? "Unreachable" : j));
}

// Test
console.log(maxJumpsToEachIndex([2, 3, 1, 1, 4])); // Output: [0, 1, 1, 2, 2]
console.log(maxJumpsToEachIndex([3, 2, 1, 0, 4])); // Output: [0, 1, 1, 1, "Unreachable"]
console.log(maxJumpsToEachIndex([1, 1, 1, 1])); // Output: [0, 1, 2, 3]
```

### Explanation:

1. Use a queue to process indices that can be reached.
2. For each index, update the maximum jumps to reach its neighbors.

3. Return the final array, replacing unreachable indices with "Unreachable".

### Time Complexity:

- **O(n^2)**: BFS processes every possible jump in the worst case.

### Space Complexity:

- **O(n)**: Space for the queue and jumps array.

## Predicting Jump Patterns in Arrays - Longest Reachable Subarray

**447. Problem: Given an array `nums`, find the longest subarray (starting from index 0) that is reachable. Return the indices of the start and end of the subarray.**

### Solution:

```
function longestReachableSubarray(nums) {
  let farthest = 0,
    start = 0,
    end = 0;

  for (let i = 0; i < nums.length; i++) {
    if (i > farthest) break; // Stop if the current index is unreachable
    farthest = Math.max(farthest, i + nums[i]);
    end = Math.min(farthest, nums.length - 1); // Update the end of the
subarray
  }

  return [start, end];
}

// Test
console.log(longestReachableSubarray([2, 3, 1, 1, 4])); // Output: [0, 4]
console.log(longestReachableSubarray([3, 2, 1, 0, 4])); // Output: [0, 3]
console.log(longestReachableSubarray([1, 1, 1, 1])); // Output: [0, 3]
console.log(longestReachableSubarray([0])); // Output: [0, 0]
```

### Explanation:

1. Track the farthest index reachable as you iterate through the array.
2. If the current index exceeds the farthest reachable index, stop and return the subarray indices.
3. The subarray `[start, end]` represents the longest contiguous sequence of reachable indices.

### Time Complexity:

- **O(n)**: Single pass through the array.

### Space Complexity:

- **O(1)**: No additional space used.

## Predicting Jump Patterns in Arrays - Count All Possible Paths

**448. Problem: Count all possible ways to jump from index 0 to the last index of the array. A jump is valid if it adheres to the `nums[i]` constraint.**

### Solution:

```
function countPaths(nums) {
  const n = nums.length;
  const memo = new Array(n).fill(-1);

  function dfs(index) {
    if (index === n - 1) return 1; // Base case: reached the last index
    if (memo[index] !== -1) return memo[index];

    let ways = 0;
    for (let i = 1; i <= nums[index]; i++) {
      if (index + i < n) {
        ways += dfs(index + i);
      }
    }

    memo[index] = ways;
    return ways;
  }

  return dfs(0);
}

// Test
console.log(countPaths([2, 3, 1, 1, 4])); // Output: 5
console.log(countPaths([3, 2, 1, 0, 4])); // Output: 0
console.log(countPaths([1, 1, 1, 1])); // Output: 6
console.log(countPaths([2, 1, 3, 1, 1])); // Output: 4
```

### Explanation:

1. Use a recursive function `dfs(index)` to count the number of ways to jump from the current index to the end.
2. Use memoization to store the results for each index and avoid recalculations.

### Time Complexity:

- **O( $n^2$ ):** Each index processes its neighbors in the worst case.

### Space Complexity:

- **O( $n$ ):** Space for the memoization array and recursion stack.

## Predicting Jump Patterns in Arrays - Maximum Sum of Reachable Indices

**449. Problem:** Find the maximum sum of values for any path from index 0 to the last index of the array, following the jump constraints in `nums`.

### Solution:

```
function maxSumJump(nums) {
  const n = nums.length;
  const dp = Array(n).fill(-Infinity);
  dp[0] = nums[0];

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

  return dp[n - 1];
}

// Test
console.log(maxSumJump([2, 3, 1, 1, 4])); // Output: 10
console.log(maxSumJump([3, 2, 1, 0, 4])); // Output: -Infinity (cannot reach last index)
console.log(maxSumJump([1, 2, 3, 4, 5])); // Output: 15
console.log(maxSumJump([10, -10, 5, 2, 1])); // Output: 13
```

### Explanation:

1. Use the `dp` array to store the maximum sum to reach each index.
2. For each index `i`, check all previous indices `j` that can jump to `i` and update `dp[i]`.
3. Return `dp[n-1]`, the maximum sum to reach the last index.

### Time Complexity:

- **O( $n^2$ ):** Nested loops to calculate transitions for each index.

### Space Complexity:

- **O(n)**: Space for the DP array.

## Predicting Jump Patterns in Arrays - Minimum Cost to Reach Last Index

**450. Problem:** Given an array `nums`, where each element represents the cost of stepping on that index, find the minimum cost to reach the last index starting from index 0.

**Solution:**

```
function minCostJump(nums) {
  const n = nums.length;
  const dp = Array(n).fill(Infinity);
  dp[0] = nums[0];

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

  return dp[n - 1];
}

// Test
console.log(minCostJump([1, 3, 1, 1, 4])); // Output: 5
console.log(minCostJump([2, 2, 2, 2, 2])); // Output: 8
console.log(minCostJump([10, 1, 2, 1, 1])); // Output: 13
console.log(minCostJump([1, 0, 1, 1, 100])); // Output: 3
```

**Explanation:**

1. Use the `dp` array to store the minimum cost to reach each index.
2. For each index `i`, check all previous indices `j` that can jump to `i` and update `dp[i]`.
3. Return `dp[n-1]`, the minimum cost to reach the last index.

**Time Complexity:**

- **O(n^2)**: Nested loops to calculate transitions for each index.

**Space Complexity:**

- **O(n)**: Space for the DP array.

## 5. Linked List Operations

### Reversing Elements in a Linked List

**451. Problem:** Given the head of a singly linked list, reverse the list and return its new head.

Input: head = [1, 2, 3, 4, 5];  
Output: [5, 4, 3, 2, 1];

Input: head = [1, 2];  
Output: [2, 1];

Input: head = [];  
Output: [];

**Solution:**

```
function reverseList(head) {  
    let prev = null;  
    let current = head;  
  
    while (current) {  
        const next = current.next; // Store the next node  
        current.next = prev; // Reverse the current node's pointer  
        prev = current; // Move prev to the current node  
        current = next; // Move to the next node  
    }  
  
    return prev;  
}  
  
// ListNode class  
class ListNode {  
    constructor(val = 0, next = null) {  
        this.val = val;  
        this.next = next;  
    }
}  
  
// Helper function to create a linked list from an array  
function createLinkedList(arr) {  
    let head = null;  
    let tail = null;  
  
    for (const val of arr) {  
        const newNode = new ListNode(val);  
        if (!head) {  
            head = newNode;  
            tail = head;
    }
}
```

```

} else {
    tail.next = newNode;
    tail = newNode;
}
}

return head;
}

// Helper function to convert a linked list to an array
function linkedListToArray(head) {
    const result = [];
    while (head) {
        result.push(head.val);
        head = head.next;
    }
    return result;
}

// Test
const head = createLinkedList([1, 2, 3, 4, 5]);
const reversed = reverseList(head);
console.log(linkedListToArray(reversed)); // Output: [5, 4, 3, 2, 1]

```

### Explanation:

#### 1. Iterative Approach:

- Use a while loop to traverse the list.
- Reverse the next pointer of each node to point to the previous node.
- Maintain three pointers (prev, current, next) to achieve this efficiently.

#### 2. Recursive Approach:

- Base case: If the current node is null or the last node, return it as the new head.
- Recursively reverse the rest of the list and adjust the pointers.

### Time Complexity:

- **O(n):** Iterate through all nodes.

### Space Complexity:

- **Iterative: O(1)** (constant space).
- **Recursive: O(n)** (space for recursion stack).

## Reversing Elements in a Linked List in Groups of K

**452. Problem:** Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list. If the number of nodes is not a multiple of k, leave the remaining nodes as they are.

Input: (head = [1, 2, 3, 4, 5]), (k = 2);  
Output: [2, 1, 4, 3, 5];

Input: (head = [1, 2, 3, 4, 5]), (k = 3);  
Output: [3, 2, 1, 4, 5];

### Solution:

```
function reverseKGroup(head, k) {  
    if (!head || k === 1) return head;  
  
    // Helper function to reverse a segment of the list  
    function reverse(start, end) {  
        let prev = null;  
        let current = start;  
        while (current !== end) {  
            const next = current.next;  
            current.next = prev;  
            prev = current;  
            current = next;  
        }  
        return prev;  
    }  
  
    let dummy = new ListNode(0);  
    dummy.next = head;  
    let prevGroupEnd = dummy;  
  
    while (true) {  
        let count = 0;  
        let groupStart = prevGroupEnd.next;  
        let groupEnd = groupStart;  
  
        // Move groupEnd k steps forward  
        while (groupEnd && count < k) {  
            groupEnd = groupEnd.next;  
            count++;  
        }  
  
        if (count < k) break; // If fewer than k nodes are left, stop  
  
        // Reverse the current group  
        const newStart = reverse(groupStart, groupEnd);  
        prevGroupEnd.next = newStart; // Connect the previous group to the new start  
    }  
}
```

```

        groupStart.next = groupEnd; // Connect the reversed group to the next part
        prevGroupEnd = groupStart; // Move to the end of the reversed group
    }

    return dummy.next;
}

// Test
const head = createLinkedList([1, 2, 3, 4, 5]);
const reversedK = reverseKGroup(head, 2);
console.log(linkedListToArray(reversedK)); // Output: [2, 1, 4, 3, 5]

const head2 = createLinkedList([1, 2, 3, 4, 5]);
const reversedK2 = reverseKGroup(head2, 3);
console.log(linkedListToArray(reversedK2)); // Output: [3, 2, 1, 4, 5]

```

### Explanation:

1. Traverse the list to identify groups of k nodes.
2. Reverse each group of k nodes using a helper function.
3. Use a dummy node to simplify handling of the head pointer during reversal.
4. If fewer than k nodes are left at the end, leave them unchanged.

### Time Complexity:

- **O(n):** Each node is visited once.

### Space Complexity:

- **O(1):** Constant space for pointer manipulation.

## Reverse Alternate K Nodes in a Linked List

**453. Problem:** Given the head of a linked list, reverse every alternate group of k nodes in the list, starting with the first group. Leave the other groups as they are.

Input: (head = [1, 2, 3, 4, 5, 6, 7, 8]), (k = 2);  
Output: [2, 1, 3, 4, 6, 5, 7, 8];

Input: (head = [1, 2, 3, 4, 5]), (k = 3);  
Output: [3, 2, 1, 4, 5];

### Solution:

```

function reverseAlternateKGroup(head, k) {
    if (!head || k === 1) return head;

    // Helper function to reverse a segment of the list

```

```

function reverse(start, end) {
    let prev = null;
    let current = start;
    while (current !== end) {
        const next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    return prev;
}

let dummy = new ListNode(0);
dummy.next = head;
let prevGroupEnd = dummy;
let reverseFlag = true;

while (true) {
    let count = 0;
    let groupStart = prevGroupEnd.next;
    let groupEnd = groupStart;

    // Move groupEnd k steps forward
    while (groupEnd && count < k) {
        groupEnd = groupEnd.next;
        count++;
    }

    if (count < k) break; // If fewer than k nodes are left, stop

    if (reverseFlag) {
        // Reverse the current group
        const newStart = reverse(groupStart, groupEnd);
        prevGroupEnd.next = newStart;
        groupStart.next = groupEnd;
        prevGroupEnd = groupStart;
    } else {
        // Skip this group
        prevGroupEnd = groupStart;
        for (let i = 0; i < k; i++) {
            prevGroupEnd = prevGroupEnd.next;
        }
    }

    reverseFlag = !reverseFlag; // Alternate the reversal flag
}

return dummy.next;

```

```

}

// Test
const head = createLinkedList([1, 2, 3, 4, 5, 6, 7, 8]);
const reversedAltK = reverseAlternateKGroup(head, 2);
console.log(linkedListToArray(reversedAltK)); // Output: [2, 1, 3, 4, 6, 5, 7, 8]

const head2 = createLinkedList([1, 2, 3, 4, 5]);
const reversedAltK2 = reverseAlternateKGroup(head2, 3);
console.log(linkedListToArray(reversedAltK2)); // Output: [3, 2, 1, 4, 5]

```

### Explanation:

1. Use a flag (reverseFlag) to decide whether to reverse the current group.
2. Reverse nodes in the current group if the flag is true.
3. Skip nodes in the current group if the flag is false.

### Time Complexity:

- **O(n):** Each node is visited once.

### Space Complexity:

- **O(1):** Constant space for pointer manipulation.

### Reversing Elements in a Linked List Between Two Indices

**454. Problem:** Given the head of a linked list, reverse the nodes of the list from position left to right. Positions are 1-based. Return the modified linked list.

Input: (head = [1, 2, 3, 4, 5]), (left = 2), (right = 4);  
Output: [1, 4, 3, 2, 5];

Input: (head = [1, 2, 3]), (left = 1), (right = 2);  
Output: [2, 1, 3];

### Solution:

```

function reverseBetween(head, left, right) {
  if (!head || left === right) return head;

  let dummy = new ListNode(0);
  dummy.next = head;
  let prev = dummy;

  // Move prev to the node before the left position
  for (let i = 1; i < left; i++) {
    prev = prev.next;
  }

```

```

}

// Reverse the nodes between left and right
let current = prev.next;
let next = null;
for (let i = 0; i < right - left; i++) {
    next = current.next;
    current.next = next.next;
    next.next = prev.next;
    prev.next = next;
}

return dummy.next;
}

// Test
const head = createLinkedList([1, 2, 3, 4, 5]);
const reversedPartial = reverseBetween(head, 2, 4);
console.log(linkedListToArray(reversedPartial)); // Output: [1, 4, 3, 2, 5]

```

### **Explanation:**

1. Traverse the list until the node before the left position.
2. Reverse the segment between left and right using pointer manipulation.
3. Reconnect the reversed segment with the rest of the list.

### **Time Complexity:**

- **O(n):** Traverse the list once.

### **Space Complexity:**

- **O(1):** Constant space for pointer manipulation.

## **Reverse Nodes in K-Group Recursively**

**455. Problem:** Given a linked list, reverse the nodes of the list k at a time using recursion. Return the modified list.

### **Solution:**

```

function reverseKGroupRecursive(head, k) {
    let count = 0;
    let current = head;

    // Check if there are at least k nodes to reverse
    while (current && count < k) {
        current = current.next;
    }
}

```

```

        count++;
    }

    if (count === k) {
        // Reverse the first k nodes
        const newHead = reverseKNodes(head, k);
        // Recursively reverse the rest of the list
        head.next = reverseKGroupRecursive(current, k);
        return newHead;
    }

    return head; // If fewer than k nodes are left, return the head as is
}

// Helper function to reverse k nodes
function reverseKNodes(head, k) {
    let prev = null;
    let current = head;

    while (k > 0) {
        const next = current.next;
        current.next = prev;
        prev = current;
        current = next;
        k--;
    }

    return prev;
}

// Test
const headRecursive = createLinkedList([1, 2, 3, 4, 5]);
const reversedKRecursive = reverseKGroupRecursive(headRecursive, 3);
console.log(linkedListToArray(reversedKRecursive)); // Output: [3, 2, 1, 4, 5]

```

### **Explanation:**

1. Check if there are at least  $k$  nodes to reverse.
2. Reverse the first  $k$  nodes, then recursively call the function for the rest of the list.
3. Connect the reversed segment to the result of the recursive call.

### **Time Complexity:**

- **O(n):** Each node is visited once.

### **Space Complexity:**

- **O(n/k):** Space for the recursion stack.

## Reverse a Circular Linked List

**456. Problem: Reverse a circular linked list and return the new head of the list.**

**Solution:**

```
function reverseCircularList(head) {
    if (!head || !head.next || head.next === head) return head;

    let prev = null;
    let current = head;
    const tail = head;

    // Reverse the list
    do {
        const next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    } while (current !== tail);

    // Reconnect the tail to the new head
    head.next = prev;
    return prev;
}

// Test
const circularHead = createLinkedList([1, 2, 3, 4, 5]);
circularHead.next.next.next.next = circularHead; // Make it circular
const reversedCircular = reverseCircularList(circularHead);
console.log(linkedListToArray(reversedCircular).slice(0, 5)); // Output: [5, 4, 3, 2, 1]
```

**Explanation:**

1. Use a do-while loop to traverse and reverse the circular list.
2. Reconnect the last node to the new head to restore the circular structure.

**Time Complexity:**

- **O(n):** Traverse the list once.

**Space Complexity:**

- **O(1):** Constant space for pointer manipulation.

## Reverse a Doubly Linked List

**457. Problem:** Given the head of a doubly linked list, reverse the list and return its new head.

Input: head = [1, 2, 3, 4, 5];  
Output: [5, 4, 3, 2, 1];

**Solution:**

```
function reverseDoublyLinkedList(head) {  
    if (!head || !head.next) return head;  
  
    let current = head;  
    let newHead = null;  
  
    while (current) {  
        // Swap the next and prev pointers  
        const temp = current.next;  
        current.next = current.prev;  
        current.prev = temp;  
  
        newHead = current; // Update the new head  
        current = temp; // Move to the next node  
    }  
  
    return newHead;  
}  
  
// Doubly linked list node class  
class DoublyListNode {  
    constructor(val = 0, prev = null, next = null) {  
        this.val = val;  
        this.prev = prev;  
        this.next = next;  
    }  
}  
  
// Helper function to create a doubly linked list from an array  
function createDoublyLinkedList(arr) {  
    let head = null;  
    let tail = null;  
  
    for (const val of arr) {  
        const newNode = new DoublyListNode(val);  
        if (!head) {  
            head = newNode;  
            tail = head;  
        } else {  
            tail.next = newNode;  
            newNode.prev = tail;  
            tail = newNode;  
        }  
    }  
    return head;  
}
```

```

        }
    }

    return head;
}

// Helper function to convert a doubly linked list to an array
function doublyLinkedListToArray(head) {
    const result = [];
    while (head) {
        result.push(head.val);
        head = head.next;
    }
    return result;
}

// Test
const head = createDoublyLinkedList([1, 2, 3, 4, 5]);
const reversed = reverseDoublyLinkedList(head);
console.log(doublyLinkedListToArray(reversed)); // Output: [5, 4, 3, 2, 1]

```

### **Explanation:**

1. Traverse the list and for each node, swap the next and prev pointers.
2. Update the newHead to the last node during the traversal.
3. Return the newHead as the new head of the reversed list.

### **Time Complexity:**

- **O(n):** Traverse the list once.

### **Space Complexity:**

- **O(1):** Constant space for pointer manipulation.

### **Reverse N Nodes from the Start**

**458. Problem: Given the head of a singly linked list, reverse the first n nodes of the list. Leave the rest of the list unchanged.**

Input: (head = [1, 2, 3, 4, 5]), (n = 3);  
Output: [3, 2, 1, 4, 5];

### **Solution:**

```

function reverseNNodes(head, n) {
    if (!head || n <= 1) return head;
}

```

```

let prev = null;
let current = head;

for (let i = 0; i < n && current; i++) {
    const next = current.next;
    current.next = prev;
    prev = current;
    current = next;
}

head.next = current; // Connect the reversed part to the rest of the list
return prev;
}

// Test
const headN = createLinkedList([1, 2, 3, 4, 5]);
const reversedN = reverseNNodes(headN, 3);
console.log(linkedListToArray(reversedN)); // Output: [3, 2, 1, 4, 5]

```

### **Explanation:**

1. Reverse the first n nodes using a loop.
2. Connect the last node of the reversed segment to the remaining list.
3. Return the new head of the reversed segment.

### **Time Complexity:**

- **O(n):** Reverse the first n nodes.

### **Space Complexity:**

- **O(1):** Constant space for pointer manipulation.

## **Reverse the Entire List Twice**

**459. Problem: Reverse a singly linked list twice and return the final list. The output should be the same as the input list.**

### **Solution:**

```

function reverseTwice(head) {
    if (!head || !head.next) return head;

    // Helper function to reverse a list
    function reverse(head) {
        let prev = null;
        let current = head;

```

```

while (current) {
    const next = current.next;
    current.next = prev;
    prev = current;
    current = next;
}
return prev;
}

// Reverse the list twice
const reversedOnce = reverse(head);
return reverse(reversedOnce);
}

// Test
const headTwice = createLinkedList([1, 2, 3, 4, 5]);
const reversedTwice = reverseTwice(headTwice);
console.log(linkedListToArray(reversedTwice)); // Output: [1, 2, 3, 4, 5]

```

### Explanation:

1. Reverse the list once using the pointer reversal technique.
2. Reverse it again to restore the original order.
3. Verify that the output matches the input.

### Time Complexity:

- **O(n):** Two passes through the list.

### Space Complexity:

- **O(1):** Constant space for pointer manipulation.

## Reverse K Nodes in Reverse Order

### 460. Problem: Reverse the list in groups of k but start reversing from the end of the list.

#### Solution:

```

function reverseKFromEnd(head, k) {
    if (!head || k <= 1) return head;

    const length = getLength(head);
    let dummy = new ListNode(0);
    dummy.next = head;

    let prevGroupEnd = dummy;

```

```

for (let i = 0; i < Math.floor(length / k); i++) {
    let current = prevGroupEnd.next;
    let prev = null;

    for (let j = 0; j < k; j++) {
        const next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }

    const temp = prevGroupEnd.next;
    prevGroupEnd.next = prev;
    temp.next = current;
    prevGroupEnd = temp;
}

return dummy.next;
}

// Helper function to calculate the length of the list
function getLength(head) {
    let length = 0;
    while (head) {
        length++;
        head = head.next;
    }
    return length;
}

// Test
const headKFromEnd = createLinkedList([1, 2, 3, 4, 5]);
const reversedKFromEnd = reverseKFromEnd(headKFromEnd, 2);
console.log(linkedListToArray(reversedKFromEnd)); // Output: [2, 1, 4, 3, 5]

```

### Detecting Cycles in Linked Lists

**461. Problem:** Given the head of a linked list, determine if the linked list contains a cycle. If there is a cycle, return true. Otherwise, return false.

Input: head = [3, 2, 0, -4] (tail connects to the second node)  
Output: true

Input: head = [1, 2] (tail connects to the first node)  
Output: true

Input: head = [1] (no cycle)  
Output: false

**Solution:**

```
function hasCycle(head) {
    let slow = head;
    let fast = head;

    while (fast && fast.next) {
        slow = slow.next; // Move slow by 1 step
        fast = fast.next.next; // Move fast by 2 steps
        if (slow === fast) return true; // Cycle detected
    }

    return false; // No cycle
}

// Test
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

// Helper function to create a linked list with a cycle
function createLinkedListWithCycle(values, pos) {
    let head = null,
        tail = null,
        cycleNode = null;

    values.forEach((value, index) => {
        const newNode = new ListNode(value);
        if (!head) {
            head = newNode;
            tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        if (index === pos) cycleNode = newNode;
    });

    if (tail && cycleNode) tail.next = cycleNode;

    return head;
}

// Test Cases
```

```

const head1 = createLinkedListWithCycle([3, 2, 0, -4], 1);
console.log(hasCycle(head1)); // Output: true

const head2 = createLinkedListWithCycle([1, 2], 0);
console.log(hasCycle(head2)); // Output: true

const head3 = createLinkedListWithCycle([1], -1);
console.log(hasCycle(head3)); // Output: false

```

### Explanation:

1. Use two pointers (slow and fast) to traverse the linked list.
2. If there's a cycle, the fast pointer will eventually catch up to the slow pointer.
3. If fast reaches the end of the list (null), it confirms there's no cycle.

### Time Complexity:

- **O(n):** Each pointer traverses the list at most once.

### Space Complexity:

- **O(1):** Constant space.

## Detect and Find the Starting Node of the Cycle

**462. Problem: If a cycle exists in a linked list, return the starting node of the cycle. Otherwise, return null.**

### Solution:

```

function detectCycle(head) {
    let slow = head;
    let fast = head;

    // Detect if a cycle exists
    while (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;

        if (slow === fast) {
            // Cycle detected, find the starting node
            slow = head;
            while (slow !== fast) {
                slow = slow.next;
                fast = fast.next;
            }
        }
    }
}

```

```

        return slow; // Starting node of the cycle
    }
}

return null; // No cycle
}

// Test
const headWithCycle = createLinkedListWithCycle([3, 2, 0, -4], 1);
console.log(detectCycle(headWithCycle).val); // Output: 2

const headWithoutCycle = createLinkedListWithCycle([1, 2], -1);
console.log(detectCycle(headWithoutCycle)); // Output: null

```

### **Explanation:**

1. Detect the cycle using the two-pointer technique.
2. When the cycle is detected, reset one pointer to the head and move both pointers one step at a time.
3. The point where the two pointers meet is the starting node of the cycle.

### **Time Complexity:**

- **O(n):** Detecting the cycle and finding the starting node both take linear time.

### **Space Complexity:**

- **O(1):** Constant space.

### **Length of the Cycle in a Linked List**

**463. Problem: If a cycle exists in a linked list, find the length of the cycle. If there is no cycle, return 0.**

### **Solution:**

```

function cycleLength(head) {
    let slow = head;
    let fast = head;

    // Detect if a cycle exists
    while (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;

        if (slow === fast) {
            // Cycle detected, calculate its length
            let length = 0;

```

```

        do {
            slow = slow.next;
            length++;
        } while (slow !== fast);
        return length;
    }
}

return 0; // No cycle
}

// Test
const headWithCycle = createLinkedListWithCycle([3, 2, 0, -4], 1);
console.log(cycleLength(headWithCycle)); // Output: 3

const headWithoutCycle = createLinkedListWithCycle([1, 2], -1);
console.log(cycleLength(headWithoutCycle)); // Output: 0

```

### **Explanation:**

1. Detect the cycle using the two-pointer technique.
2. Once a cycle is detected, traverse the cycle to calculate its length.
3. Return the cycle length, or 0 if there is no cycle.

### **Time Complexity:**

- **O(n):** Detecting the cycle and calculating its length both take linear time.

### **Space Complexity:**

- **O(1):** Constant space.

### **Remove the Cycle in a Linked List**

**464. Problem: If a cycle exists in a linked list, remove the cycle by setting the next pointer of the last node in the cycle to null.**

### **Solution:**

```

function removeCycle(head) {
    let slow = head;
    let fast = head;

    // Detect if a cycle exists
    while (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;
    }
}

```

```

if (slow === fast) {
    // Find the start of the cycle
    slow = head;
    let prev = null;
    while (slow !== fast) {
        prev = fast;
        slow = slow.next;
        fast = fast.next;
    }

    // Remove the cycle
    while (fast.next !== slow) {
        fast = fast.next;
    }
    fast.next = null; // Break the cycle
    return head;
}

return head; // No cycle
}

// Test
const headWithCycle = createLinkedListWithCycle([3, 2, 0, -4], 1);
removeCycle(headWithCycle);
console.log(hasCycle(headWithCycle)); // Output: false

```

### **Explanation:**

1. Detect the cycle using the two-pointer technique.
2. Find the starting node of the cycle by resetting one pointer to the head and moving both pointers one step at a time.
3. Traverse the cycle and set the next pointer of the last node to null to remove the cycle.

### **Time Complexity:**

- **O(n):** Detecting and removing the cycle both take linear time.

### **Space Complexity:**

- **O(1):** Constant space.

### **Find the First Node in the Cycle Using HashSet**

**465. Problem: Find the first node in the cycle of a linked list using a Set to track visited nodes.**

### **Solution:**

```

function detectCycleWithSet(head) {
  const visited = new Set();

  while (head) {
    if (visited.has(head)) {
      return head; // Cycle detected, return the starting node
    }
    visited.add(head);
    head = head.next;
  }

  return null; // No cycle
}

// Test
const headWithCycle = createLinkedListWithCycle([3, 2, 0, -4], 1);
console.log(detectCycleWithSet(headWithCycle).val); // Output: 2

const headWithoutCycle = createLinkedListWithCycle([1, 2], -1);
console.log(detectCycleWithSet(headWithoutCycle)); // Output: null

```

### **Explanation:**

1. Use a Set to track visited nodes.
2. Traverse the list and check if a node is already in the Set.
3. If a cycle is detected, return the starting node. Otherwise, return null.

### **Time Complexity:**

- **O(n):** Traverse the list once.

### **Space Complexity:**

- **O(n):** Space for the Set.

### **Check if a Given Node is Part of a Cycle**

**466. Problem: Given the head of a linked list and a specific node, determine if the given node is part of a cycle.**

### **Solution:**

```

function isNodeInCycle(head, targetNode) {
  let slow = head;
  let fast = head;

  // Detect if a cycle exists
  while (fast && fast.next) {

```

```

slow = slow.next;
fast = fast.next.next;

if (slow === fast) {
    // Cycle detected, traverse the cycle
    do {
        if (slow === targetNode) return true;
        slow = slow.next;
    } while (slow !== fast);

    return false; // Node is not part of the cycle
}

return false; // No cycle
}

// Test
const head = createLinkedListWithCycle([3, 2, 0, -4], 1);
const targetNode = head.next; // Node with value 2
console.log(isNodeInCycle(head, targetNode)); // Output: true

const nonCycleNode = head.next.next.next; // Node with value -4
console.log(isNodeInCycle(head, nonCycleNode)); // Output: true

```

### Explanation:

1. Detect the cycle using the two-pointer technique.
2. If a cycle exists, traverse it and compare each node in the cycle with the target node.
3. Return true if the target node is found in the cycle; otherwise, return false.

### Time Complexity:

- **O(n):** Detect the cycle and check the nodes in the cycle.

### Space Complexity:

- **O(1):** Constant space for pointer manipulation.

### Find the Middle Node in a Cycle

**467. Problem: If a cycle exists in a linked list, find the middle node of the cycle.**

### Solution:

```

function findMiddleOfCycle(head) {
    let slow = head;
    let fast = head;

```

```

// Detect if a cycle exists
while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;

    if (slow === fast) {
        // Cycle detected, find its length
        let length = 0;
        do {
            slow = slow.next;
            length++;
        } while (slow !== fast);

        // Find the middle node
        slow = fast;
        for (let i = 0; i < Math.floor(length / 2); i++) {
            slow = slow.next;
        }
        return slow;
    }
}

return null; // No cycle
}

// Test
const headWithCycle = createLinkedListWithCycle([3, 2, 0, -4], 1);
console.log(findMiddleOfCycle(headWithCycle).val); // Output: 0 (middle node of the cycle)

```

### **Explanation:**

1. Detect the cycle and calculate its length.
2. Traverse half the length of the cycle to reach the middle node.

### **Time Complexity:**

- **O(n):** Detect the cycle and find its middle node.

### **Space Complexity:**

- **O(1):** Constant space.

### **Merge Two Linked Lists and Check for Cycles**

**468. Problem:** Given two singly linked lists, merge them into one list. If a cycle exists in either of the original lists, ensure that it is retained in the merged list.

**Solution:**

```
function mergeListsAndCheckCycles(head1, head2) {
    if (!head1) return head2;
    if (!head2) return head1;

    let tail1 = head1;
    while (tail1.next) {
        tail1 = tail1.next;
    }

    // Check if the first list has a cycle
    let hasCycle1 = hasCycle(head1);
    let hasCycle2 = hasCycle(head2);

    // Merge the lists
    tail1.next = head2;

    // Restore the cycle if it exists
    if (hasCycle1 || hasCycle2) {
        let cycleStart1 = detectCycle(head1);
        let cycleStart2 = detectCycle(head2);

        if (hasCycle1) {
            tail1.next = cycleStart1;
        } else if (hasCycle2) {
            tail1.next = cycleStart2;
        }
    }
}

return head1;
}

// Test
const head1 = createLinkedListWithCycle([1, 2, 3], 1);
const head2 = createLinkedListWithCycle([4, 5, 6], 0);
const mergedHead = mergeListsAndCheckCycles(head1, head2);
console.log(hasCycle(mergedHead)); // Output: true
console.log(detectCycle(mergedHead).val); // Output: 2 (start of the cycle)
```

**Explanation:**

1. Merge the two lists by connecting the tail of the first list to the head of the second list.
2. Retain any cycles in the original lists by reconnecting the tail node to the appropriate cycle start node.

**Time Complexity:**

- **O(n + m):** Traverse both lists to merge and check for cycles.

### Space Complexity:

- **O(1):** Constant space for pointer manipulation.

### Count Nodes in a Cycle of a Linked List

**469. Problem: If a cycle exists in a linked list, count the number of nodes in the cycle. Return 0 if there is no cycle.**

#### Solution:

```
function countCycleNodes(head) {
  let slow = head;
  let fast = head;

  // Detect if a cycle exists
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;

    if (slow === fast) {
      // Cycle detected, count the nodes
      let count = 1;
      let current = slow.next;
      while (current !== slow) {
        count++;
        current = current.next;
      }
      return count;
    }
  }

  return 0; // No cycle
}

// Test
const headWithCycle = createLinkedListWithCycle([1, 2, 3, 4, 5], 1);
console.log(countCycleNodes(headWithCycle)); // Output: 4

const headWithoutCycle = createLinkedListWithCycle([1, 2, 3], -1);
console.log(countCycleNodes(headWithoutCycle)); // Output: 0
```

#### Explanation:

1. Detect the cycle using the two-pointer method.

2. Use one pointer to traverse the cycle and count the number of nodes until the pointer returns to the starting node.

### Time Complexity:

- **O(n)**: Detect and count nodes in the cycle.

### Space Complexity:

- **O(1)**: Constant space.

## Convert a Cyclic Linked List into an Acyclic List

**470. Problem: Given a cyclic linked list, break the cycle by setting the next pointer of the last node in the cycle to null.**

### Solution:

```
function breakCycle(head) {  
    let slow = head;  
    let fast = head;  
  
    // Detect if a cycle exists  
    while (fast && fast.next) {  
        slow = slow.next;  
        fast = fast.next.next;  
  
        if (slow === fast) {  
            // Cycle detected, find the last node in the cycle  
            let lastNode = slow;  
            while (lastNode.next !== slow) {  
                lastNode = lastNode.next;  
            }  
            lastNode.next = null; // Break the cycle  
            return head;  
        }  
    }  
  
    return head; // No cycle  
}  
  
// Test  
const headWithCycle = createLinkedListWithCycle([1, 2, 3, 4, 5], 1);  
breakCycle(headWithCycle);  
console.log(hasCycle(headWithCycle)); // Output: false  
console.log(linkedListToArray(headWithCycle)); // Output: [1, 2, 3, 4, 5]
```

### Explanation:

1. Detect the cycle using the two-pointer technique.
2. Find the last node in the cycle and set its next pointer to null to break the cycle.

### Time Complexity:

- **O(n)**: Detect and break the cycle.

### Space Complexity:

- **O(1)**: Constant space.

## Merging Two Ordered Linked Lists

**471. Problem:** You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted linked list and return its head.

Input: (list1 = [1, 2, 4]), (list2 = [1, 3, 4]);  
 Output: [1, 1, 2, 3, 4, 4];

Input: (list1 = []), (list2 = []);  
 Output: [];

Input: (list1 = []), (list2 = [0]);  
 Output: [0];

### Solution:

```
function mergeTwoLists(list1, list2) {
  const dummy = new ListNode(0);
  let current = dummy;

  while (list1 && list2) {
    if (list1.val <= list2.val) {
      current.next = list1;
      list1 = list1.next;
    } else {
      current.next = list2;
      list2 = list2.next;
    }
    current = current.next;
  }

  // Append the remaining nodes of the non-empty list
  current.next = list1 || list2;

  return dummy.next;
}
```

```

// ListNode class
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

// Helper function to create a linked list from an array
function createLinkedList(arr) {
    let head = null;
    let tail = null;

    for (const val of arr) {
        const newNode = new ListNode(val);
        if (!head) {
            head = newNode;
            tail = head;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
    }

    return head;
}

// Helper function to convert a linked list to an array
function linkedListToArray(head) {
    const result = [];
    while (head) {
        result.push(head.val);
        head = head.next;
    }
    return result;
}

// Test
const list1 = createLinkedList([1, 2, 4]);
const list2 = createLinkedList([1, 3, 4]);
const mergedList = mergeTwoLists(list1, list2);
console.log(linkedListToArray(mergedList)); // Output: [1, 1, 2, 3, 4, 4]

```

### Explanation:

1. Use a dummy node to simplify the process of building the merged list.
2. Compare the values of the nodes in list1 and list2 and append the smaller node to the merged list.

3. After reaching the end of one list, append the remaining nodes of the other list.
4. Return dummy.next as the head of the merged list.

### Time Complexity:

- **O(n + m):** Where n and m are the lengths of list1 and list2.

### Space Complexity:

- **O(1):** The merging is done in place without using extra space.

## Merging Two Ordered Linked Lists Recursively

**472. Problem: You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted linked list using recursion and return its head.**

### Solution:

```
function mergeTwoListsRecursive(list1, list2) {
  if (!list1) return list2; // If list1 is empty, return list2
  if (!list2) return list1;

  if (list1.val <= list2.val) {
    list1.next = mergeTwoListsRecursive(list1.next, list2);
    return list1;
  } else {
    list2.next = mergeTwoListsRecursive(list1, list2.next);
    return list2;
  }
}

// Test
const list1 = createLinkedList([1, 2, 4]);
const list2 = createLinkedList([1, 3, 4]);
const mergedListRecursive = mergeTwoListsRecursive(list1, list2);
console.log(linkedListToArray(mergedListRecursive)); // Output: [1, 1, 2, 3, 4, 4]
```

### Explanation:

1. Check if either list is empty; if so, return the non-empty list as the merged list.
2. Compare the values of the current nodes of list1 and list2:
  - If list1's value is smaller or equal, set its next to the result of merging the remainder of list1 and list2.
  - Otherwise, set list2's next to the result of merging list1 and the remainder of list2.
3. The recursion ensures that the merged list is built in sorted order.

### Time Complexity:

- **O(n + m):** Where n and m are the lengths of list1 and list2.

### Space Complexity:

- **O(n + m):** Recursion stack space for the length of the lists.

### Merging Two Ordered Linked Lists into a Circular Linked List

**473. Problem: Merge two sorted linked lists into a single sorted circular linked list. Return the head of the new circular list.**

#### Solution:

```
function mergeTwoCircularLists(list1, list2) {
  const dummy = new ListNode(0);
  let current = dummy;

  let head1 = list1;
  let head2 = list2;

  // Merge the two lists
  while (head1 && head2) {
    if (head1.val <= head2.val) {
      current.next = head1;
      head1 = head1.next;
    } else {
      current.next = head2;
      head2 = head2.next;
    }
    current = current.next;
  }

  // Append the remaining nodes
  current.next = head1 || head2;

  // Find the tail of the merged list and connect it to the head
  while (current.next) {
    current = current.next;
  }
  current.next = dummy.next; // Connect tail to head

  return dummy.next;
}

// Test
const circularList1 = createLinkedList([1, 3, 5]);
circularList1.next.next.next = circularList1; // Make it circular

const circularList2 = createLinkedList([2, 4, 6]);
```

```

circularList2.next.next.next = circularList2; // Make it circular

const mergedCircularList = mergeTwoCircularLists(circularList1, circularList2);
console.log(linkedListToArray(mergedCircularList).slice(0, 6)); // Output: [1, 2, 3, 4, 5, 6]

```

### **Explanation:**

1. Merge the two lists as usual.
2. After merging, traverse to the tail of the merged list and connect it back to the head to form a circular list.
3. Return the head of the new circular list.

### **Time Complexity:**

- **O(n + m):** Traverse both lists to merge and make circular.

### **Space Complexity:**

- **O(1):** In-place merging.

## **Merging Two Ordered Linked Lists Using a Priority Queue**

**474. Problem: Merge two sorted linked lists using a priority queue to ensure the merged list is sorted.**

### **Solution:**

```

function mergeTwoListsWithPriorityQueue(list1, list2) {
  const heap = [];
  const dummy = new ListNode(0);
  let current = dummy;

  if (list1) heap.push(list1);
  if (list2) heap.push(list2);

  heap.sort((a, b) => a.val - b.val); // Min-heap using array sort

  while (heap.length > 0) {
    const smallest = heap.shift(); // Extract the smallest node
    current.next = smallest;
    current = current.next;

    if (smallest.next) {
      heap.push(smallest.next);
      heap.sort((a, b) => a.val - b.val); // Re-sort the heap
    }
  }
}

```

```

    return dummy.next;
}

// Test
const list1 = createLinkedList([1, 2, 4]);
const list2 = createLinkedList([1, 3, 4]);
const mergedWithHeap = mergeTwoListsWithPriorityQueue(list1, list2);
console.log(linkedListToArray(mergedWithHeap)); // Output: [1, 1, 2, 3, 4, 4]

```

### Explanation:

1. Use a min-heap to efficiently keep track of the smallest element across both lists.
2. Extract the smallest node and append it to the merged list.
3. Add the next node from the same list to the heap and repeat until both lists are fully traversed.

### Time Complexity:

- $O((n + m) * \log(n + m))$ : Inserting and extracting nodes from the heap.

### Space Complexity:

- $O(n + m)$ : Space for the heap.

## Merge K Sorted Linked Lists

**475. Problem:** Given an array of k sorted linked lists, merge them into one sorted linked list and return its head.

### Solution:

```

function mergeKLists(lists) {
  if (!lists.length) return null;

  // Helper function to merge two lists
  function mergeTwoLists(list1, list2) {
    const dummy = new ListNode(0);
    let current = dummy;

    while (list1 && list2) {
      if (list1.val <= list2.val) {
        current.next = list1;
        list1 = list1.next;
      } else {
        current.next = list2;
        list2 = list2.next;
      }
      current = current.next;
    }

    return dummy.next;
  }

  return mergeTwoLists(...lists);
}

```

```

    }

    current.next = list1 || list2;
    return dummy.next;
}

while (lists.length > 1) {
  const mergedLists = [];

  for (let i = 0; i < lists.length; i += 2) {
    const list1 = lists[i];
    const list2 = i + 1 < lists.length ? lists[i + 1] : null;
    mergedLists.push(mergeTwoLists(list1, list2));
  }

  lists = mergedLists;
}

return lists[0];
}

// Test
const list1 = createLinkedList([1, 4, 5]);
const list2 = createLinkedList([1, 3, 4]);
const list3 = createLinkedList([2, 6]);
const mergedK = mergeKLists([list1, list2, list3]);
console.log(linkedListToArray(mergedK)); // Output: [1, 1, 2, 3, 4, 4, 5, 6]

```

### Explanation:

#### 1. Divide and Conquer Approach:

- Repeatedly merge pairs of linked lists until one list remains.
- This approach has a time complexity of  $O(N \log k)$ , where  $N$  is the total number of nodes and  $k$  is the number of lists.

#### 2. Priority Queue Approach:

- Use a min-heap to always extract the smallest node among all lists.
- Insert the next node of the extracted node into the heap to maintain the order.
- This approach has a time complexity of  $O(N \log k)$  as well, but with a slightly higher overhead due to heap operations.

### Space Complexity for Both Approaches:

- $O(k)$ : For maintaining the heap in the priority queue approach or the merged list pairs in the divide and conquer approach.

### Merging Two Ordered Linked Lists Alternately

**476. Problem: Merge two sorted linked lists alternately, maintaining the order within each list. If one list is longer, append the remaining nodes to the merged list.**

**Solution:**

```
function mergeListsAlternately(list1, list2) {  
    const dummy = new ListNode(0);  
    let current = dummy;  
  
    while (list1 && list2) {  
        current.next = list1;  
        list1 = list1.next;  
        current = current.next;  
  
        current.next = list2;  
        list2 = list2.next;  
        current = current.next;  
    }  
  
    // Append the remaining nodes  
    current.next = list1 || list2;  
  
    return dummy.next;  
}  
  
// Test  
const list1 = createLinkedList([1, 3, 5]);  
const list2 = createLinkedList([2, 4, 6, 8]);  
const mergedAlternately = mergeListsAlternately(list1, list2);  
console.log(linkedListToArray(mergedAlternately)); // Output: [1, 2, 3, 4, 5, 6, 8]
```

**Explanation:**

1. Append a node from list1 to the merged list, then append a node from list2.
2. If one list is longer, append its remaining nodes after finishing alternation.

**Time Complexity:**

- **O(n + m):** Traverse both lists once.

**Space Complexity:**

- **O(1):** In-place merging.

### **Merging Two Ordered Linked Lists Without Dummy Node**

**477. Problem: Merge two sorted linked lists without using a dummy node.**

**Solution:**

```
function mergeTwoListsNoDummy(list1, list2) {  
    if (!list1) return list2; // If list1 is empty, return list2  
    if (!list2) return list1; // If list2 is empty, return list1  
  
    // Determine the head of the merged list  
    let head;  
    if (list1.val <= list2.val) {  
        head = list1;  
        list1 = list1.next;  
    } else {  
        head = list2;  
        list2 = list2.next;  
    }  
  
    let current = head;  
  
    // Merge the two lists  
    while (list1 && list2) {  
        if (list1.val <= list2.val) {  
            current.next = list1;  
            list1 = list1.next;  
        } else {  
            current.next = list2;  
            list2 = list2.next;  
        }  
        current = current.next;  
    }  
  
    // Append the remaining nodes  
    current.next = list1 || list2;  
  
    return head;  
}  
  
// Test  
const list1 = createLinkedList([1, 2, 4]);  
const list2 = createLinkedList([1, 3, 4]);  
const mergedListNoDummy = mergeTwoListsNoDummy(list1, list2);  
console.log(linkedListToArray(mergedListNoDummy)); // Output: [1, 1, 2, 3, 4, 4]
```

**Explanation:**

1. Check for empty lists and handle them directly.
2. Set the head of the merged list based on the smaller value between the heads of list1 and list2.
3. Use a pointer to merge the two lists in place without using a dummy node.

### Time Complexity:

- **O(n + m)**: Traverse both lists once.

### Space Complexity:

- **O(1)**: In-place merging.

## Merging Two Ordered Linked Lists in Reverse Order

**478. Problem: Merge two sorted linked lists into one sorted list, but in reverse order.**

### Solution:

```
function mergeTwoListsReverse(list1, list2) {  
    let head = null;  
  
    while (list1 && list2) {  
        if (list1.val <= list2.val) {  
            const temp = list1.next;  
            list1.next = head;  
            head = list1;  
            list1 = temp;  
        } else {  
            const temp = list2.next;  
            list2.next = head;  
            head = list2;  
            list2 = temp;  
        }  
    }  
  
    while (list1) {  
        const temp = list1.next;  
        list1.next = head;  
        head = list1;  
        list1 = temp;  
    }  
  
    while (list2) {  
        const temp = list2.next;  
        list2.next = head;  
        head = list2;  
        list2 = temp;  
    }  
  
    return head;  
}  
  
// Test
```

```
const list1 = createLinkedList([1, 2, 4]);
const list2 = createLinkedList([1, 3, 4]);
const mergedListReverse = mergeTwoListsReverse(list1, list2);
console.log(linkedListToArray(mergedListReverse)); // Output: [4, 4, 3, 2, 1, 1]
```

### Explanation:

1. Traverse both lists and prepend the smaller node to the merged list.
2. When one list is fully traversed, prepend the remaining nodes from the other list to the result.
3. The result is a merged list sorted in reverse order.

### Time Complexity:

- **O(n + m)**: Traverse both lists once.

### Space Complexity:

- **O(1)**: In-place merging.

## Merging Two Ordered Linked Lists Without Modifying Input

**479. Problem: Merge two sorted linked lists into a new sorted linked list without modifying the input lists.**

### Solution:

```
function mergeTwoListsCopy(list1, list2) {
  const dummy = new ListNode(0);
  let current = dummy;

  while (list1 && list2) {
    if (list1.val <= list2.val) {
      current.next = new ListNode(list1.val);
      list1 = list1.next;
    } else {
      current.next = new ListNode(list2.val);
      list2 = list2.next;
    }
    current = current.next;
  }

  while (list1) {
    current.next = new ListNode(list1.val);
    list1 = list1.next;
    current = current.next;
  }
}
```

```

while (list2) {
    current.next = new ListNode(list2.val);
    list2 = list2.next;
    current = current.next;
}

return dummy.next;
}

// Test
const list1 = createLinkedList([1, 2, 4]);
const list2 = createLinkedList([1, 3, 4]);
const mergedListCopy = mergeTwoListsCopy(list1, list2);
console.log(linkedListToArray(mergedListCopy)); // Output: [1, 1, 2, 3, 4, 4]

```

### **Explanation:**

1. Create a new linked list using a dummy node.
2. Traverse both input lists and copy the smaller node into the new list.
3. When one list is fully traversed, copy the remaining nodes of the other list.

### **Time Complexity:**

- **O(n + m):** Traverse both lists once.

### **Space Complexity:**

- **O(n + m):** New list space for the merged result.

## **Merging Two Ordered Linked Lists (Recursive Approach Without Modifying Input)**

### **480. Problem:**

Given two sorted linked lists, merge them into a single sorted linked list **without modifying the input lists**, using a **recursive approach**. Instead of modifying the existing nodes, create a new linked list and return its head.

### **Example 1:**

Input:  
List 1: 1 → 3 → 5  
List 2: 2 → 4 → 6

Output:  
Merged List: 1 → 2 → 3 → 4 → 5 → 6

### Solution:

```
class ListNode {  
    constructor(val = 0, next = null) {  
        this.val = val;  
        this.next = next;  
    }  
}  
  
function mergeTwoSortedListsRecursive(l1, l2) {  
    // Base case: if one of the lists is empty, return the other  
    if (!l1) return l2;  
    if (!l2) return l1;  
  
    // Create a new node with the smaller value and recurse  
    if (l1.val < l2.val) {  
        return new ListNode(l1.val, mergeTwoSortedListsRecursive(l1.next, l2));  
    } else {  
        return new ListNode(l2.val, mergeTwoSortedListsRecursive(l1, l2.next));  
    }  
}
```

### Explanation:

#### Objective:

Merge two sorted linked lists into a single sorted list **without modifying the input lists** using recursion.

#### Approach:

1. We use **recursion** to build a new linked list while maintaining sorted order.
2. Each recursive call processes one node and passes the rest of the lists for merging.

#### Steps in Detail:

1. **Base Case:**
  - o If either l1 or l2 is null, return the other list.
  - o This ensures that the non-empty list is appended automatically when the other list is fully processed.
2. **Recursive Comparison:**
  - o Compare the values of the first nodes of both lists.
  - o Select the smaller node and create a **new node** with that value.
  - o Recursively call the function to merge the remaining nodes.
3. **Constructing the New List:**
  - o The new node's next pointer is set to the result of the recursive call.
  - o This process continues until both lists are fully processed.
4. **Returning the Merged List:**
  - o At the end of recursion, we return the **new merged list**.

## Time and Space Complexity:

- **Time Complexity:  $O(n + m)$** 
  - We traverse both lists fully once.
- **Space Complexity:  $O(n + m)$** 
  - Because of recursive function calls, extra space is used on the call stack.

## Merging Multiple Sorted Data Sets (Using Min Heap - Priority Queue)

### 481: Problem:

You are given an array of  $k$  linked lists, where each linked list is **sorted in ascending order**. Your task is to merge all  $k$  linked lists into **one sorted linked list** and return its head.

#### Input:

```
lists = [
    1 → 4 → 5,
    1 → 3 → 4,
    2 → 6
]
```

#### Output:

```
1 → 1 → 2 → 3 → 4 → 4 → 5 → 6
```

#### Solution:

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

class MinHeap {
    constructor() {
        this.heap = [];
    }

    push(node) {
        this.heap.push(node);
        this.heap.sort((a, b) => a.val - b.val);
    }

    pop() {
        return this.heap.shift();
    }
}
```

```

size() {
    return this.heap.length;
}
}

function mergeKSortedLists(lists) {
    let minHeap = new MinHeap();

    // Add all list heads to the heap
    for (let list of lists) {
        if (list) minHeap.push(list);
    }

    let dummy = new ListNode(-1);
    let current = dummy;

    while (minHeap.size() > 0) {
        let smallestNode = minHeap.pop();
        current.next = smallestNode;
        current = current.next;

        if (smallestNode.next) {
            minHeap.push(smallestNode.next);
        }
    }

    return dummy.next;
}

```

### **Explanation:**

### **Objective:**

Efficiently merge k sorted linked lists into one sorted list.

### **Approach:**

- We use a **Min Heap (Priority Queue)** to always extract the smallest element from the k lists.
- Since each list is already sorted, we only need to compare the head nodes of each list.

### **Steps:**

1. **Initialize a Min Heap**
  - The Min Heap is used to keep track of the smallest element among the k lists.
  - Initially, we insert the head of each list into the heap.
2. **Extract the smallest element and build the merged list**
  - Remove the smallest element from the heap.

- Add it to the new merged list.
  - Move to the next node in the corresponding list and push it into the heap.
3. **Continue until all elements are processed.**
- The process continues until the heap is empty.

### Why it Works:

- Since the heap always maintains the smallest element at the top, we can efficiently build a sorted linked list.
- The priority queue ensures that at each step, we select the smallest available node.

### Time Complexity:

- **$O(N \log K)$** , where N is the total number of nodes and K is the number of lists.
- Each node is inserted and removed from the heap, which takes  **$O(\log K)$**  time.

### Space Complexity:

- **$O(K)$**  for the heap storage, where K is the number of linked lists.

## Merging Multiple Sorted Data Sets (Using Divide and Conquer)

### 482. Problem:

You are given an array of k linked lists, where each linked list is **sorted in ascending order**. Your task is to merge all k sorted linked lists into **one sorted linked list** and return its head. You need to implement the solution using the **divide and conquer** approach.

#### Example 1:

##### Input:

```
lists = [
    1 → 4 → 5,
    1 → 3 → 4,
    2 → 6
]
```

##### Output:

```
1 → 1 → 2 → 3 → 4 → 4 → 5 → 6
```

### Solution:

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}
```

```

        this.next = next;
    }
}

function mergeTwoLists(l1, l2) {
    let dummy = new ListNode(-1);
    let current = dummy;

    while (l1 && l2) {
        if (l1.val < l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    current.next = l1 || l2;
    return dummy.next;
}

function mergeKSortedLists(lists) {
    if (lists.length === 0) return null;

    while (lists.length > 1) {
        let mergedLists = [];

        for (let i = 0; i < lists.length; i += 2) {
            let l1 = lists[i];
            let l2 = lists[i + 1] || null; // Handle the case when there is an odd number of lists
            mergedLists.push(mergeTwoLists(l1, l2));
        }

        lists = mergedLists; // Update the lists array with the merged lists
    }

    return lists[0];
}

```

### **Explanation:**

#### **Objective:**

Merge k sorted linked lists into one sorted list using a **divide and conquer** approach.

#### **Approach:**

### 1. Merging Two Lists:

We begin by defining a helper function `mergeTwoLists()` that merges two sorted lists. The function compares the nodes of both lists and adds the smaller node to the merged list.

### 2. Divide and Conquer:

- Initially, we have  $k$  lists in the array. We pair these lists and merge them using the helper function `mergeTwoLists()`.
- After each round of merging, we reduce the number of lists by half. This process continues until only one list remains.

### 3. Handling Odd Number of Lists:

If there is an odd number of lists in a round, we handle it by simply merging the last list with null.

### 4. Final Result:

Once the list is reduced to a single list, it will be the merged result of all the input lists.

## Why it Works:

- The divide and conquer approach ensures that we are always working with smaller and smaller subproblems.
- At each step, we reduce the size of the problem by half, merging pairs of lists in each iteration.

## Time Complexity:

- The time complexity is  $O(N \log K)$ , where  $N$  is the total number of nodes across all lists, and  $K$  is the number of lists.
- Each merge operation takes  $O(N)$  time, and the merge process reduces the number of lists by half at each step, requiring  $\log K$  iterations.

## Space Complexity:

- $O(K)$  space for storing the input lists and merged results at each stage.

## Merging Multiple Sorted Data Sets (Using Min Heap / Priority Queue)

### 483. Problem:

You are given an array of  $k$  linked lists, where each linked list is **sorted in ascending order**. Your task is to merge all  $k$  sorted linked lists into **one sorted linked list** and return its head. This time, solve it using a **Min Heap (Priority Queue)**.

### Example 1:

#### Input:

```
lists = [
    1 → 4 → 5,
    1 → 3 → 4,
```

```
2 → 6  
]
```

### Output:

```
1 → 1 → 2 → 3 → 4 → 4 → 5 → 6
```

### Solution:

```
class ListNode {  
    constructor(val = 0, next = null) {  
        this.val = val;  
        this.next = next;  
    }  
}  
  
class MinHeap {  
    constructor() {  
        this.heap = [];  
    }  
  
    insert(node) {  
        this.heap.push(node);  
        this.heapifyUp();  
    }  
  
    extractMin() {  
        if (this.heap.length === 1) return this.heap.pop();  
        const min = this.heap[0];  
        this.heap[0] = this.heap.pop();  
        this.heapifyDown();  
        return min;  
    }  
  
    heapifyUp() {  
        let index = this.heap.length - 1;  
        while (index > 0) {  
            let parentIndex = Math.floor((index - 1) / 2);  
            if (this.heap[parentIndex].val <= this.heap[index].val) break;  
            [this.heap[parentIndex], this.heap[index]] = [  
                this.heap[index],  
                this.heap[parentIndex],  
            ];  
            index = parentIndex;  
        }  
    }  
}
```

```

heapifyDown() {
    let index = 0;
    const length = this.heap.length;
    while (true) {
        let left = 2 * index + 1;
        let right = 2 * index + 2;
        let smallest = index;

        if (left < length && this.heap[left].val < this.heap[smallest].val)
            smallest = left;
        if (right < length && this.heap[right].val < this.heap[smallest].val)
            smallest = right;

        if (smallest === index) break;

        [this.heap[index], this.heap[smallest]] = [
            this.heap[smallest],
            this.heap[index],
        ];
        index = smallest;
    }
}

isEmpty() {
    return this.heap.length === 0;
}
}

function mergeKSortedLists(lists) {
    let minHeap = new MinHeap();

    // Insert all list heads into the minHeap
    for (let list of lists) {
        if (list) minHeap.insert(list);
    }

    let dummy = new ListNode(-1);
    let current = dummy;

    // Process heap until empty
    while (!minHeap.isEmpty()) {
        let minNode = minHeap.extractMin();
        current.next = minNode;
        current = current.next;

        if (minNode.next) {
            minHeap.insert(minNode.next);
        }
    }
}

```

```

    }
}

return dummy.next;
}

```

### Explanation:

### Objective:

Merge k sorted linked lists into one sorted list using a **Min Heap (Priority Queue)**.

### Approach:

#### 1. Using a Min Heap:

- The Min Heap (Priority Queue) helps efficiently find the smallest node among all lists.
- Instead of sorting entire lists, we extract **only the smallest** available node at each step.

#### 2. Inserting Initial Nodes:

- We start by inserting the **head nodes** of all k lists into the Min Heap.
- This allows us to always keep track of the smallest element.

#### 3. Building the Merged List:

- We **extract the smallest node** from the heap and add it to our result list.
- If the extracted node has a next node, we **insert** it into the heap.
- This ensures that we always process the smallest available value.

#### 4. Continue Until the Heap is Empty:

- Once the heap is empty, all nodes have been processed, and the merged list is complete.

### Why it Works:

- The Min Heap allows us to efficiently maintain the order by always extracting the **smallest** available node.
- The process ensures that we **only compare necessary elements**, making it highly optimized.

### Time Complexity:

- **$O(N \log K)$** , where N is the total number of nodes, and K is the number of lists.
- Inserting and extracting from a Min Heap takes  **$O(\log K)$**  time.
- Since we process **N nodes**, the total time complexity is  **$O(N \log K)$** .

### Space Complexity:

- **$O(K)$**  for the Min Heap (storing at most K elements at a time).
- **$O(1)$**  additional space apart from the result list.

## Merging Multiple Sorted Data Sets (Using Divide and Conquer Approach)

### 484. Problem:

Given k sorted linked lists, merge them into **one sorted linked list** and return the head.  
This time, solve it using a **Divide and Conquer approach**.

#### Example 1:

##### Input:

```
lists = [
    1 → 4 → 5,
    1 → 3 → 4,
    2 → 6
]
```

##### Output:

1 → 1 → 2 → 3 → 4 → 4 → 5 → 6

#### Solution:

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

// Function to merge two sorted linked lists
function mergeTwoLists(l1, l2) {
    let dummy = new ListNode(-1);
    let current = dummy;

    while (l1 !== null && l2 !== null) {
        if (l1.val < l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    if (l1 !== null) current.next = l1;
    if (l2 !== null) current.next = l2;

    return dummy.next;
}
```

```

// Function to merge k sorted linked lists using Divide and Conquer
function mergeKSortedLists(lists) {
    if (!lists.length) return null;

    while (lists.length > 1) {
        let mergedLists = [];

        for (let i = 0; i < lists.length; i += 2) {
            let l1 = lists[i];
            let l2 = i + 1 < lists.length ? lists[i + 1] : null;
            mergedLists.push(mergeTwoLists(l1, l2));
        }

        lists = mergedLists;
    }

    return lists[0];
}

```

### Explanation:

#### Objective:

Merge k sorted linked lists into one sorted list using **Divide and Conquer**.

#### Approach:

1. **Divide and Conquer Strategy:**
  - o Instead of merging all lists at once, we **merge pairs of lists** at each step.
  - o This reduces the total number of lists at each iteration.
2. **Merging Pairs of Lists:**
  - o We use the mergeTwoLists() function to **merge two sorted linked lists** efficiently.
  - o After merging, we store the new merged lists in an array.
3. **Repeat Until One List Remains:**
  - o We keep repeating the process until we are left with just **one final sorted list**.

#### Why This Works Efficiently?

- This approach reduces the problem size at each step, similar to **Merge Sort**.
- Instead of performing  $k-1$  sequential merges, we **halve** the number of lists at each iteration.
- **Time Complexity:**  $O(N \log k)$ , where  $N$  is the total number of nodes.

#### Merging Multiple Sorted Data Sets (Using Min Heap/Priority Queue Approach)

#### 485. Problem:

Given k sorted linked lists, merge them into **one sorted linked list** and return the head.  
This time, solve it using a **Min Heap (Priority Queue) approach**.

### Example 1:

#### Input:

```
lists = [
    1 → 4 → 5,
    1 → 3 → 4,
    2 → 6
]
```

#### Output:

1 → 1 → 2 → 3 → 4 → 4 → 5 → 6

#### Solution:

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

class MinHeap {
    constructor() {
        this.heap = [];
    }

    insert(node) {
        this.heap.push(node);
        this.bubbleUp();
    }

    bubbleUp() {
        let index = this.heap.length - 1;
        while (index > 0) {
            let parentIndex = Math.floor((index - 1) / 2);
            if (this.heap[parentIndex].val <= this.heap[index].val) break;
            [this.heap[parentIndex], this.heap[index]] = [
                this.heap[index],
                this.heap[parentIndex],
            ];
            index = parentIndex;
        }
    }
}
```

```

extractMin() {
    if (this.heap.length === 1) return this.heap.pop();
    let min = this.heap[0];
    this.heap[0] = this.heap.pop();
    this.sinkDown(0);
    return min;
}

sinkDown(index) {
    let left = 2 * index + 1,
        right = 2 * index + 2;
    let smallest = index;

    if (
        left < this.heap.length &&
        this.heap[left].val < this.heap[smallest].val
    ) {
        smallest = left;
    }
    if (
        right < this.heap.length &&
        this.heap[right].val < this.heap[smallest].val
    ) {
        smallest = right;
    }
    if (smallest !== index) {
        [this.heap[smallest], this.heap[index]] = [
            this.heap[index],
            this.heap[smallest],
        ];
        this.sinkDown(smallest);
    }
}

isEmpty() {
    return this.heap.length === 0;
}

// Function to merge k sorted linked lists using Min Heap
function mergeKSortedLists(lists) {
    let minHeap = new MinHeap();

    // Insert the first node of each list into the min heap
    for (let list of lists) {
        if (list) minHeap.insert(list);
    }
}

```

```

let dummy = new ListNode(-1);
let current = dummy;

while (!minHeap.isEmpty()) {
    let smallest = minHeap.extractMin();
    current.next = smallest;
    current = current.next;

    if (smallest.next) {
        minHeap.insert(smallest.next);
    }
}

return dummy.next;
}

```

### **Explanation:**

### **Objective:**

Merge k sorted linked lists into one sorted list using a **Min Heap (Priority Queue)** for efficient extraction of the smallest element.

### **Approach:**

1. **Using a Min Heap (Priority Queue):**
  - o The Min Heap helps **extract the smallest element** among multiple sorted lists in **O(log k)** time.
  - o The heap always maintains the smallest value at the top.
2. **Insert Initial Elements:**
  - o Add the first node of each linked list into the Min Heap.
3. **Iteratively Extract Minimum and Add to Result List:**
  - o Extract the smallest node from the heap.
  - o Attach it to the result list.
  - o Insert the next node from the extracted node's list into the heap (if available).
  - o Repeat until the heap is empty.

### **Why This Works Efficiently?**

- **Min Heap ensures that the smallest node is always processed first.**
- Instead of scanning all lists at each step, we **only deal with k elements in the heap** at a time.
- **Time Complexity:**  $O(N \log k)$ , where N is the total number of nodes and k is the number of lists.
- **Space Complexity:**  $O(k)$  for the heap storage.

### **Merging Multiple Sorted Data Sets (Using Divide and Conquer Approach)**

**486. Problem:** Given k sorted linked lists, merge them into one sorted linked list and return the head.

This time, solve it using a Divide and Conquer approach.

**Example 1:**

**Input:**

```
lists = [
    1 → 4 → 5,
    1 → 3 → 4,
    2 → 6
]
```

**Output:**

```
1 → 1 → 2 → 3 → 4 → 4 → 5 → 6
```

**Solution:**

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

// Merging Two Ordered Linked Lists
function mergeTwoLists(l1, l2) {
    let dummy = new ListNode(-1);
    let current = dummy;

    while (l1 !== null && l2 !== null) {
        if (l1.val < l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    if (l1 !== null) current.next = l1;
    if (l2 !== null) current.next = l2;

    return dummy.next;
}
```

```

}

// Merging Multiple Sorted Data Sets using Divide and Conquer
function mergeKLists(lists) {
    if (!lists || lists.length === 0) return null;

    while (lists.length > 1) {
        let mergedLists = [];

        for (let i = 0; i < lists.length; i += 2) {
            let l1 = lists[i];
            let l2 = i + 1 < lists.length ? lists[i + 1] : null;
            mergedLists.push(mergeTwoLists(l1, l2));
        }

        lists = mergedLists;
    }

    return lists[0];
}

```

### Explanation:

### Objective:

Merge k sorted linked lists using the **Divide and Conquer approach**, reducing the number of lists in **each iteration**.

### Approach:

#### 1. Pairwise Merging:

- We merge two lists at a time, reducing the total number of lists by half in each step.
- This is **similar to the merge step of Merge Sort**.

#### 2. Continue Until One List Remains:

- We repeat the merging process until only **one final sorted list** remains.

#### 3. Merging Two Sorted Lists:

- A helper function `mergeTwoLists(l1, l2)` is used to merge two linked lists in  **$O(n)$  time**.
- It works like the **merge step in Merge Sort**.

### Why This Works Efficiently?

- Instead of repeatedly merging **from the beginning**, we merge lists **in pairs**, reducing the number of merges needed.
- **Time Complexity:  $O(N \log k)$** 
  - Each merge step takes  **$O(N)$**  time.
  - Since we **divide the number of lists by 2** in each iteration, there are  **$O(\log k)$  levels** of merging.

- This results in an efficient  **$O(N \log k)$  complexity**.
- **Space Complexity:  $O(1)$** 
  - We merge lists in place without extra memory.

## Merging Multiple Sorted Data Sets (Using Min Heap / Priority Queue Approach)

### 487. Problem:

Given  $k$  sorted linked lists, merge them into **one sorted linked list** and return the head. This time, solve it using a **Min Heap (Priority Queue) approach**.

#### Example 1:

##### Input:

```
lists = [
  1 → 4 → 5,
  1 → 3 → 4,
  2 → 6
]
```

##### Output:

```
1 → 1 → 2 → 3 → 4 → 4 → 5 → 6
```

#### Solution:

```
class ListNode {
  constructor(val = 0, next = null) {
    this.val = val;
    this.next = next;
  }
}

// Min Heap implementation using JavaScript Priority Queue
class MinHeap {
  constructor() {
    this.heap = [];
  }

  // Insert a node into the heap
  insert(node) {
    if (node) {
      this.heap.push(node);
      this.heap.sort((a, b) => a.val - b.val);
    }
  }

  // Extract the smallest node from the heap
}
```

```

extractMin() {
    return this.heap.shift();
}

// Check if the heap is empty
isEmpty() {
    return this.heap.length === 0;
}

// Merging Multiple Sorted Data Sets using Min Heap
function mergeKLists(lists) {
    if (!lists || lists.length === 0) return null;

    let minHeap = new MinHeap();

    // Insert the first node of each list into the heap
    for (let list of lists) {
        if (list) minHeap.insert(list);
    }

    let dummy = new ListNode(-1);
    let current = dummy;

    // Extract the smallest element and insert the next node from the same list
    while (!minHeap.isEmpty()) {
        let minNode = minHeap.extractMin();
        current.next = minNode;
        current = current.next;

        if (minNode.next) {
            minHeap.insert(minNode.next);
        }
    }

    return dummy.next;
}

```

### Explanation:

#### Objective:

Merge k sorted linked lists using a **Min Heap** for efficient selection of the smallest element.

#### Approach:

##### 1. Use a Min Heap (Priority Queue)

- Insert the first node of each list into the heap.
- Extract the **smallest node** and append it to the merged list.

- Insert the **next node** from the extracted node's list back into the heap.
- Repeat until all nodes are processed.

## 2. Why Use a Min Heap?

- The Min Heap ensures that we always extract the **smallest available element** in  **$O(\log k)$  time**.
- Instead of comparing all elements manually, the heap maintains the **sorted order efficiently**.

### Why This Works Efficiently?

- **Insertion into Heap:**  $O(\log k)$  (heapify after insertion)
- **Extracting the Min Node:**  $O(\log k)$  (heapify after removal)
- **Total Operations:**  $O(N \log k)$ 
  - N is the total number of nodes.
  - k is the number of linked lists.

### Time Complexity: $O(N \log k)$

- We insert and remove each of the N nodes from the heap **log k times**, leading to  **$O(N \log k)$** .

### Space Complexity: $O(k)$

- The heap stores at most k nodes at any time.

### Removing Specific Nodes in a Sequence (Two-Pointer Approach)

**488. Problem:** Given the head of a linked list, remove the nth node from the end of the list and return the modified linked list.

#### Example 1:

##### Input:

```
(head = [1, 2, 3, 4, 5]), (n = 2);
```

##### Output:

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

#### Solution:

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}
```

```

function removeNthFromEnd(head, n) {
    let dummy = new ListNode(0);
    dummy.next = head;
    let first = dummy;
    let second = dummy;

    // Move the first pointer n + 1 steps ahead
    for (let i = 0; i <= n; i++) {
        first = first.next;
    }

    // Move both pointers until first reaches the end
    while (first !== null) {
        first = first.next;
        second = second.next;
    }

    // Skip the target node
    second.next = second.next.next;

    return dummy.next;
}

```

### **Explanation:**

### **Objective:**

Remove the nth node from the end of a linked list in **one pass (O(N) time complexity)**.

### **Approach:**

1. **Create a dummy node** before the head to handle edge cases.
2. **Use two pointers:**
  - o Move first pointer  $n + 1$  steps ahead.
  - o Keep second pointer at the dummy node.
3. **Move both pointers until first reaches the end.**
4. **Now, second.next points to the node to remove.**
5. **Skip the node by updating second.next.**

### **Why This Works Efficiently?**

- **Only one traversal of the linked list (O(N)).**
- **No extra space required (O(1)).**
- **Handles edge cases (like removing the first node) using a dummy node.**

## Removing Specific Nodes in a Sequence (Using Stack Approach)

**489. Problem:** Given the head of a linked list, remove the nth node from the end of the list and return the modified linked list.

**Example 1:**

**Input:**

```
head = [10, 20, 30, 40, 50], (n = 3);
```

**Output:**

```
[10, 20, 40, 50];
```

**Solution:**

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

function removeNthFromEnd(head, n) {
    let stack = [];
    let dummy = new ListNode(0);
    dummy.next = head;
    let current = dummy;

    // Push all nodes onto the stack
    while (current) {
        stack.push(current);
        current = current.next;
    }

    // Pop `n` nodes
    for (let i = 0; i < n; i++) {
        stack.pop();
    }

    // Get the previous node
    let prevNode = stack.pop();
    prevNode.next = prevNode.next.next;

    return dummy.next;
}
```

**Explanation:**

**Objective:**

Remove the nth node from the end using a **Stack ( $O(N)$  time complexity)**.

### Approach:

1. **Create a dummy node** before the head to handle edge cases.
2. **Push all nodes into a stack.**
3. **Pop n nodes** to reach the node before the one we need to remove.
4. **Update the next pointer** of the previous node.
5. **Return the modified linked list.**

### Why This Works Efficiently?

- **Single pass to build the stack ( $O(N)$ ).**
- **Another pass to remove the node ( $O(N)$ ).**
- **Uses extra space ( $O(N)$ ) for the stack.**
- **No need to compute the length explicitly.**

### Removing Specific Nodes in a Sequence (Two Pointers Approach)

**490. Problem:** Given the head of a linked list, remove the nth node from the end of the list and return the modified linked list.

#### Example 1:

##### Input:

```
head = [10, 20, 30, 40, 50], (n = 3);
```

##### Output:

```
[10,20,40,50]
```

##### Solution:

```
class ListNode {  
    constructor(val = 0, next = null) {  
        this.val = val;  
        this.next = next;  
    }  
}  
  
function removeNthFromEnd(head, n) {  
    let dummy = new ListNode(0);  
    dummy.next = head;  
    let fast = dummy;  
    let slow = dummy;  
  
    // Move fast pointer `n + 1` steps ahead  
    for (let i = 1; i <= n + 1; i++) {  
        fast = fast.next;  
    }
```

```

}

// Move both fast and slow pointers together
while (fast != null) {
    slow = slow.next;
    fast = fast.next;
}

// Delete the nth node from the end
slow.next = slow.next.next;

return dummy.next;
}

```

### Explanation:

### Objective:

We want to remove the nth node from the end of the linked list in **one pass** with the **two-pointer technique**.

### Approach:

- Dummy Node:** Create a dummy node at the start to handle edge cases like when the head node needs to be removed.
- Advance the Fast Pointer:** Move the fast pointer  $n + 1$  steps ahead to maintain a gap of  $n$  nodes between fast and slow.
- Move Both Pointers Together:** Move both slow and fast pointers one step at a time until fast reaches the end of the list. By then, slow will be positioned just before the node to be removed.
- Skip the Node:** Update the next pointer of the slow node to skip the nth node.
- Return the Result:** Return the modified list starting from the next of the dummy node.

### Why This Works Efficiently?

- **One pass through the list ( $O(N)$  time complexity).**
- **No extra space for auxiliary data structures ( $O(1)$  space complexity).**
- **Handles all edge cases** like removing the first or last node.

### Removing Specific Nodes in a Sequence (Using Stack)

**491. Problem:** Given the head of a linked list, remove the nth node from the end of the list and return the modified linked list.

#### Input:

(head = [10, 20, 30, 40, 50]), (n = 2);

#### Output:

```
[10, 20, 30, 50];
```

### Solution:

```
class ListNode {
    constructor(val = 0, next = null) {
        this.val = val;
        this.next = next;
    }
}

function removeNthFromEnd(head, n) {
    let stack = [];
    let current = head;

    // Push all nodes onto the stack
    while (current !== null) {
        stack.push(current);
        current = current.next;
    }

    // Pop the nth node from the stack
    for (let i = 0; i < n; i++) {
        stack.pop();
    }

    // If the stack is empty, remove the head
    if (stack.length === 0) {
        return head ? head.next : null;
    }

    // The node just before the node to remove
    let prevNode = stack[stack.length - 1];

    // Remove the nth node from the end
    prevNode.next = prevNode.next ? prevNode.next.next : null;

    return head;
}
```

### Explanation:

#### Objective:

We want to remove the nth node from the end of the linked list using a stack to keep track of nodes.

## Approach:

1. **Traverse the List:** We traverse the list and push all the nodes into a stack. Since a stack operates on the **LIFO** (Last In, First Out) principle, the last element added to the stack will be the last node of the list.
2. **Pop Nodes:** By popping  $n$  elements from the stack, we get the node before the  $n$ th node from the end. This ensures that we can correctly remove the required node.
3. **Remove the nth node:** After popping  $n$  nodes, the next node will be the one that needs to be removed. Update the next pointer to skip that node.

## Why This Works Efficiently?

- **Traversing the list and adding all nodes into the stack ensures** that we can easily access the node to be removed by popping from the stack.
- **One additional pass through the list to pop and update** is needed.
- **This method uses extra space for the stack.**

## Removing Specific Nodes in a Sequence (Two Pointers)

**492. Problem: Given the head of a linked list, remove the nth node from the end of the list and return the modified linked list.**

### Input:

```
head = [10, 20, 30, 40, 50], (n = 2);
```

### Output:

```
[10, 20, 30, 50];
```

### Solution:

```
class ListNode {  
    constructor(val = 0, next = null) {  
        this.val = val;  
        this.next = next;  
    }  
  
    function removeNthFromEnd(head, n) {  
        let dummy = new ListNode(0);  
        dummy.next = head;  
        let first = dummy;  
        let second = dummy;  
  
        // Advance first pointer by n+1 steps to get the gap  
        for (let i = 1; i <= n + 1; i++) {  
            first = first.next;  
        }
```

```

}

// Move both pointers until first reaches the end
while (first !== null) {
    first = first.next;
    second = second.next;
}

// Remove the nth node from the end
second.next = second.next.next;

return dummy.next;
}

```

### Explanation:

### Objective:

We need to remove the nth node from the end of the linked list efficiently using a two-pointer technique.

### Approach:

- Dummy Node:** A dummy node is used to handle edge cases such as removing the head node. We attach this dummy node before the head of the list.
- First Pointer:** Move the first pointer  $n+1$  steps ahead, so that the distance between the first and second pointer is  $n$  nodes.
- Simultaneous Movement:** Move both the first and second pointers one step at a time until the first pointer reaches the end of the list. At that point, the second pointer will be just before the node to be removed.
- Remove Node:** Adjust the next pointer of the second pointer to skip the node that needs to be removed.

### Why This Works Efficiently?

- Two Pointers:** By advancing the first pointer by  $n+1$  steps initially, we ensure that the second pointer will always be one step before the node to be removed when the first pointer reaches the end.
- Single Pass:** We make only one pass through the list, which is efficient.

### Optimizing Linked List Reordering

#### 493. Problem: Given a singly linked list, reorder the list so that it follows this pattern:

$L_0 \rightarrow L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n \rightarrow L_{(n-1)} \rightarrow L_{(n-2)} \rightarrow \dots \rightarrow L_1 \rightarrow L_0$

You must do this **in-place** without altering the node values.

### Example 1:

#### Input:

```
head = [1, 2, 3, 4, 5];
```

#### Output:

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

#### Explanation:

- Reorder the list to the pattern: [1, 5, 2, 4, 3].

#### Solution:

```
class ListNode {  
    constructor(val = 0, next = null) {  
        this.val = val;  
        this.next = next;  
    }  
}  
  
function reorderList(head) {  
    if (!head || !head.next) return;  
  
    // Step 1: Find the middle of the linked list  
    let slow = head;  
    let fast = head;  
  
    while (fast && fast.next) {  
        slow = slow.next;  
        fast = fast.next.next;  
    }  
  
    // Step 2: Reverse the second half  
    let second = slow.next;  
    slow.next = null;  
    second = reverseList(second);  
  
    // Step 3: Merge the two halves  
    let first = head;  
    while (second) {  
        let tmp1 = first.next;  
        let tmp2 = second.next;  
  
        first.next = second;  
        second.next = tmp1;  
    }  
}
```

```

first = tmp1;
second = tmp2;
}

function reverseList(head) {
  let prev = null;
  let curr = head;
  while (curr) {
    let nextTemp = curr.next;
    curr.next = prev;
    prev = curr;
    curr = nextTemp;
  }
  return prev;
}

```

### **Explanation:**

### **Objective:**

Reorder the list by following a specific pattern where the first node is followed by the last node, then the second node, followed by the second-last node, and so on.

### **Approach:**

#### **1. Find the Middle of the List:**

- Use two pointers: a slow pointer (which moves one step at a time) and a fast pointer (which moves two steps at a time).
- When the fast pointer reaches the end of the list, the slow pointer will be at the middle.

#### **2. Reverse the Second Half:**

- After finding the middle of the list, reverse the second half of the list. This reversal is essential because the desired order is achieved by alternately picking nodes from the first half and the reversed second half.

#### **3. Merge the Two Halves:**

- Once both halves are separated and the second half is reversed, merge the two halves by alternating nodes from the first half and the reversed second half.
- We iterate through both halves and link the nodes in the desired order.

### **Why This Works Efficiently?**

- **In-place Modification:** We reorder the list in-place, meaning no extra space is used for storing a new list. Only the pointers are modified.
- **Optimal Time Complexity:** The solution involves traversing the list twice — once to find the middle and once to merge the two halves — resulting in a time complexity of  $O(N)$ .

- **Space Complexity:** We only use a constant amount of extra space for pointers, so the space complexity is **O(1)**.

## Optimizing Linked List Reordering

### 494. Problem:

Given a singly linked list, reorder it such that:

- The first node is followed by the last node.
- Then the second node is followed by the second-last node, and so on.

For example: Given 1 -> 2 -> 3 -> 4 -> 5, reorder it to 1 -> 5 -> 2 -> 4 -> 3

### Solution:

```
function reorderList(head) {
    if (!head || !head.next || !head.next.next) return;

    // Step 1: Find the middle of the linked list using slow and fast pointers
    let slow = head;
    let fast = head;
    while (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // Step 2: Split the list into two halves
    let second = slow.next;
    slow.next = null;

    // Step 3: Reverse the second half of the list
    second = reverseList(second);

    // Step 4: Merge the two halves
    let first = head;
    while (second) {
        let tmp1 = first.next;
        let tmp2 = second.next;

        first.next = second;
        second.next = tmp1;

        first = tmp1;
        second = tmp2;
    }
}

// Helper function to reverse the linked list
```

```
function reverseList(head) {  
    let prev = null;  
    let current = head;  
    while (current) {  
        let next = current.next;  
        current.next = prev;  
        prev = current;  
        current = next;  
    }  
    return prev;  
}
```

### Explanation:

#### Step 1: Find the middle of the linked list:

- The slow pointer (slow) moves one step at a time.
- The fast pointer (fast) moves two steps at a time.
- When the fast pointer reaches the end of the list (or becomes null), the slow pointer will be at the middle node.

Example:

- Given the list 1 -> 2 -> 3 -> 4 -> 5, the slow pointer will stop at 3 (the middle node).

#### Step 2: Split the list into two halves:

- After finding the middle, we split the list at the middle. The first half will end at the slow pointer, and the second half will start at the node after the slow pointer.

Example:

- First half: 1 -> 2 -> 3
- Second half: 4 -> 5

#### Step 3: Reverse the second half:

- We reverse the second half using a helper function (reverseList) that takes the head of the second half and returns the reversed list.

Example:

- The second half 4 -> 5 becomes 5 -> 4 after reversal.

#### **Step 4: Merge the two halves:**

- We then merge the two halves by alternating between the first half and the reversed second half.
- First, we take a node from the first half (1), then from the reversed second half (5), then again from the first half (2), then from the reversed second half (4), and so on.

Example:

- The merged list becomes: 1 -> 5 -> 2 -> 4 -> 3.

#### **Time Complexity:**

- **Finding the middle:** O(N), where N is the number of nodes in the list.
- **Reversing the second half:** O(N/2), which simplifies to O(N).
- **Merging the two halves:** O(N).

Thus, the overall **Time Complexity** is **O(N)**.

#### **Space Complexity:**

- We are using constant extra space for the pointers (slow, fast, first, second, tmp1, tmp2).
- The **Space Complexity** is **O(1)**, as we are modifying the list in-place.

### **Remove Duplicates from Sorted List**

#### **495. Problem:**

Given a sorted linked list, delete all duplicates such that each element appears only once.

#### **For example:**

- Input: 1 -> 1 -> 2 -> 3 -> 3
- Output: 1 -> 2 -> 3

#### **Solution:**

```
function deleteDuplicates(head) {  
    let current = head;  
  
    while (current && current.next) {  
        if (current.val === current.next.val) {  
            current.next = current.next.next;  
        } else {  
            current = current.next;  
        }  
    }  
  
    return head;
```

```
}
```

### Explanation:

1. **Traversal of the List:**
  - o Start from the head of the linked list.
  - o Use a pointer (current) to traverse the list. For each node, compare its value with the next node's value.
2. **Check for Duplicates:**
  - o If the current node's value is equal to the next node's value, we have found a duplicate.
  - o In that case, we simply update the current.next pointer to skip over the duplicate (i.e., we point it to current.next.next).
3. **Move to the Next Node:**
  - o If the values are not equal, simply move the current pointer to the next node.
4. **End of List:**
  - o When the end of the list is reached, the loop terminates.

### Time Complexity:

- We traverse the list once, so the time complexity is **O(N)**, where N is the number of nodes in the linked list.

### Space Complexity:

- The space complexity is **O(1)** because we are not using any extra data structures, just modifying pointers in the original list.

## Swap Nodes in Pairs

### 496. Problem:

Given a linked list, swap every two adjacent nodes and return its head.

You must solve this problem in **one-pass** using **constant extra space**.

For example:

- Input: 1 -> 2 -> 3 -> 4
- Output: 2 -> 1 -> 4 -> 3

### Solution:

```
function swapPairs(head) {  
    let dummy = new ListNode(0); // Dummy node to simplify swapping head  
    dummy.next = head;  
    let current = dummy;
```

```

while (current.next && current.next.next) {
    let first = current.next;
    let second = current.next.next;

    // Perform the swap
    first.next = second.next;
    second.next = first;
    current.next = second;

    // Move the pointer forward by two nodes
    current = first;
}

return dummy.next;
}

```

### **Explanation:**

#### **1. Dummy Node:**

- We create a dummy node (dummy) with an arbitrary value. This simplifies edge cases like when the list has an odd number of elements or when the head node itself is swapped.

#### **2. Swapping in Pairs:**

- We initialize a pointer current to point to the dummy node.
- For each pair of nodes (first and second), we perform the following swaps:
  - Set first.next to second.next (this ensures that first now points to the node after the pair).
  - Set second.next to first (this makes second point to the swapped first node).
  - Update current.next to second, which points to the swapped pair.

#### **3. Move to the Next Pair:**

- After swapping the current pair, we move current to first, which is now the second node of the swapped pair. This allows us to process the next pair.

#### **4. End of List:**

- The loop terminates when there are fewer than two nodes left to swap.

### **Time Complexity:**

- The time complexity is **O(N)**, where N is the number of nodes in the list, because we only iterate through the list once.

### **Space Complexity:**

- The space complexity is **O(1)** because we are only using a constant amount of extra space, no additional data structures are required.

## **Optimizing Linked List Reordering**

### 497. Problem:

Given a singly linked list, reorder it so that:

$L_0 \rightarrow L_1 \rightarrow L_2 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$  becomes:

$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

### Solution:

```
function reorderList(head) {
    if (!head || !head.next) return;

    // Step 1: Find the middle of the list
    let slow = head,
        fast = head;
    while (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // Step 2: Reverse the second half
    let second = slow.next;
    slow.next = null;
    second = reverseList(second);

    // Step 3: Merge the two halves
    let first = head;
    while (second) {
        let temp1 = first.next;
        let temp2 = second.next;

        first.next = second;
        second.next = temp1;

        first = temp1;
        second = temp2;
    }
}

function reverseList(head) {
    let prev = null;
    while (head) {
        let next = head.next;
        head.next = prev;
        prev = head;
        head = next;
    }
}
```

```
    }
    return prev;
}
```

### Explanation:

#### 1. Finding the Middle:

- o We initialize two pointers, slow and fast. The slow pointer moves one step at a time, while fast moves two steps at a time. By the time fast reaches the end of the list, slow will be at the middle node.

#### 2. Reversing the Second Half:

- o Once we find the middle, we break the list into two halves. We reverse the second half using the reverseList helper function. This function iteratively changes the next pointers of the nodes to reverse their direction.

#### 3. Merging the Two Halves:

- o After reversing the second half, we merge the first half and the reversed second half by alternating nodes. We use two pointers (first and second), and at each step, we link first to second, and second to the next node from first.

#### 4. Edge Cases:

- o The list may be empty or contain only one node, in which case there's nothing to reorder, and we return immediately.

### Time Complexity:

- The time complexity is **O(N)**, where N is the number of nodes in the linked list. We perform a few passes through the list (finding the middle, reversing, and merging).

### Space Complexity:

- The space complexity is **O(1)**, since we only use a constant amount of extra space for the pointers (no additional data structures).

## 6. Bit Manipulation & Binary Operations

### Sum Calculation Without Arithmetic Operators (Without Using + or -)

**498. Problem:** Given two integers a and b, return the sum of the two integers without using the + or - operators.

**Example 1:**

**Input:**

```
a = 2, b = 3;
```

**Output:**

## Example 2:

### Input:

```
a = -1, b = 1;
```

### Output:

0

### Solution:

```
function getSum(a, b) {  
    while (b !== 0) {  
        let carry = (a & b) << 1; // Calculate carry  
        a = a ^ b; // Sum without carry  
        b = carry; // Assign carry to b  
    }  
    return a;  
}  
  
// Example Usage:  
console.log(getSum(2, 3)); // Output: 5  
console.log(getSum(-1, 1)); // Output: 0
```

## Explanation:

### Step 1: Understanding Bitwise Addition

- Instead of using +, we use bitwise **XOR** (^) to sum numbers without carry.
- Instead of using -, we use bitwise **AND** (&) with a left shift (<< 1) to determine the carry.

### Step 2: Iterative Approach

Initial State:

```
a = 2(10 in binary);  
b = 3(11 in binary);
```

First Iteration:

```
carry = (a & b) << 1 = (10 & 11) << 1 = 10 << 1 = 100 (4 in decimal)  
a = a ^ b = 10 ^ 11 = 01 (1 in decimal)  
b = carry = 4
```

Second Iteration:

```
carry = (a & b) << 1 = (01 & 100) << 1 = 0  
a = a ^ b = 01 ^ 100 = 101 (5 in decimal)  
b = carry = 0
```

Since b is now 0, we return a = 5.

### Why This Works

- Each iteration **adds without carry** and **computes carry separately**.
- The loop continues until there is no carry left.
- This ensures that the sum is computed correctly using only bitwise operations.

### Time & Space Complexity

- **Time Complexity: O(1)** (In most cases, it runs in constant time since integers are 32-bit.)
- **Space Complexity: O(1)** (Only a few extra variables are used.)

### Sum Calculation Without Arithmetic Operators (Recursive Approach - Without + or -)

**499. Problem:** Given two integers a and b, return their sum **without using** the + or - operators. This time, solve the problem using a **recursive approach**.

**Example 1:**

**Input:**

```
a = 4, b = 5;
```

**Output:**

9

**Example 2:**

**Input:**

```
a = -7, b = 3;
```

**Output:**

-4

**Solution:**

```

function getSum(a, b) {
  return b === 0 ? a : getSum(a ^ b, (a & b) << 1);
}

// Example Usage:
console.log(getSum(4, 5)); // Output: 9
console.log(getSum(-7, 3)); // Output: -4
console.log(getSum(0, 10)); // Output: 10
console.log(getSum(7, -3)); // Output: 4

```

## Explanation

### Step 1: Understanding Recursion with Bitwise Addition

- Instead of using `+`, we use **bitwise XOR (^)** to sum numbers without carry.
- Instead of using `-`, we use **bitwise AND (&)** and shift left (`<< 1`) to compute the carry.

### Step 2: Recursive Breakdown

Let's say  $a = 4$  (100 in binary) and  $b = 5$  (101 in binary).

#### 1. First Call: `getSum(4, 5)`

```

XOR (a ^ b) = 100 ^ 101 = 001 (1 in decimal)
Carry (a & b) = 100 & 101 = 100 (4 in decimal)
Shift Carry = 100 << 1 = 1000 (8 in decimal)
Next Call: getSum(1, 8)

```

#### Second Call: `getSum(1, 8)`

```

XOR (1 ^ 8) = 001 ^ 1000 = 1001 (9 in decimal)
Carry (1 & 8) = 001 & 1000 = 0000 (0 in decimal)
Next Call: getSum(9, 0)

```

#### Third Call: `getSum(9, 0)`

- Since  $b == 0$ , return  $a = 9$ .

## Why This Works

- Each recursive call **adds without carry** and **computes carry separately**.
- The function keeps calling itself until there is **no carry left**.
- This ensures we get the correct sum using **only bitwise operations**.

## Time & Space Complexity

- **Time Complexity: O(1)** (In most cases, recursion depth is limited to the number of bits in an integer, which is constant).
- **Space Complexity: O(1)** (Only a few extra variables are used in recursive calls).

## Sum Calculation Without Arithmetic Operators (Iterative Approach - Without + or -)

**500. Problem:** Given two integers a and b, return their sum **without using the + or - operators**. This time, solve the problem using an **iterative approach**.

**Example 1:**

**Input:**

a = 7, b = 3

**Output:**

10

**Solution:**

```
function getSum(a, b) {  
    while (b !== 0) {  
        let sumWithoutCarry = a ^ b; // XOR to get sum without carry  
        let carry = (a & b) << 1; // AND to get carry and shift left  
        a = sumWithoutCarry; // Update sum  
        b = carry; // Update carry  
    }  
    return a;  
}  
  
// Example Usage:  
console.log(getSum(7, 3)); // Output: 10  
console.log(getSum(-4, -6)); // Output: -10  
console.log(getSum(5, -2)); // Output: 3  
console.log(getSum(0, 8)); // Output: 8  
console.log(getSum(-7, 2)); // Output: -5
```

## Explanation

### Step 1: Understanding Bitwise Addition

- **XOR (^)** finds the sum **without carry**.

- **AND (&) + Left Shift (<< 1)** finds the carry and moves it to the left.

### Step 2: Iterative Breakdown

Let's take  $a = 7$  (0111 in binary) and  $b = 3$  (0011 in binary).

First Iteration:

```
XOR (7 ^ 3) = 0111 ^ 0011 = 0100 (4 in decimal)
Carry (7 & 3) = 0111 & 0011 = 0011 (3 in decimal)
Shift Carry = 0011 << 1 = 0110 (6 in decimal)
New Values: a = 4, b = 6
```

Second Iteration:

```
XOR (4 ^ 6) = 0100 ^ 0110 = 0010 (2 in decimal)
Carry (4 & 6) = 0100 & 0110 = 0100 (4 in decimal)
Shift Carry = 0100 << 1 = 1000 (8 in decimal)
New Values: a = 2, b = 8
```

Third Iteration:

```
XOR (2 ^ 8) = 0010 ^ 1000 = 1010 (10 in decimal)
Carry (2 & 8) = 0010 & 1000 = 0000 (0 in decimal)
Since b = 0, return a = 10.
```

### Why This Works

- Each iteration **adds without carry** and **computes carry separately**.
- The process **stops when no carry remains** ( $b == 0$ ).
- This ensures we get the correct sum **only using bitwise operations**.

### Time & Space Complexity

- **Time Complexity: O(1)** (In most cases, loop runs a limited number of times based on bit length).
- **Space Complexity: O(1)** (Only a few variables are used).

### Sum Calculation Without Arithmetic Operators (Recursive Approach - Without + or -)

**501. Problem:** Given two integers  $a$  and  $b$ , return their sum **without using the + or - operators**. This time, solve the problem using a **recursive approach**.

**Example 1:**

**Input:**

$a = 7, b = 3$

Output:

10

**Solution:**

```
function getSum(a, b) {  
    if (b === 0) return a; // Base case: If no carry left, return sum  
    return getSum(a ^ b, (a & b) << 1); // Recursive call with updated values  
}  
  
// Example Usage:  
console.log(getSum(7, 3)); // Output: 10  
console.log(getSum(-4, -6)); // Output: -10  
console.log(getSum(5, -2)); // Output: 3  
console.log(getSum(0, 8)); // Output: 8  
console.log(getSum(-7, 2)); // Output: -5
```

**Explanation:**

### Step 1: Understanding Recursive Bitwise Addition

- **XOR (^)** finds the sum **without carry**.
- **AND (&) + Left Shift (<< 1)** finds the carry and moves it to the left.

### Step 2: Recursive Breakdown

Let's take  $a = 7$  (0111 in binary) and  $b = 3$  (0011 in binary).

First Recursive Call:

```
XOR (7 ^ 3) = 0111 ^ 0011 = 0100 (4 in decimal)  
Carry (7 & 3) = 0111 & 0011 = 0011 (3 in decimal)  
Shift Carry = 0011 << 1 = 0110 (6 in decimal)  
Recursive Call: getSum(4, 6)
```

Second Recursive Call:

```
XOR (4 ^ 6) = 0100 ^ 0110 = 0010 (2 in decimal)  
Carry (4 & 6) = 0100 & 0110 = 0100 (4 in decimal)  
Shift Carry = 0100 << 1 = 1000 (8 in decimal)  
Recursive Call: getSum(2, 8)
```

Third Recursive Call:

```
XOR (2 ^ 8) = 0010 ^ 1000 = 1010 (10 in decimal)
```

Carry ( $2 \& 8$ ) =  $0010 \& 1000 = 0000$  (0 in decimal)

Since  $b = 0$ , return  $a = 10$ .

## Why This Works

- The recursion keeps breaking the problem into smaller parts.
- Each recursive step **adds without carry** and **computes carry separately**.
- The recursion **stops when no carry remains** ( $b == 0$ ).
- This ensures we get the correct sum **only using bitwise operations**.

## Time & Space Complexity

- **Time Complexity: O(1)** (In most cases, recursion will run a limited number of times due to the small size of integers).
- **Space Complexity: O(1)** (No extra space used except function call stack).

## Sum Calculation Without Arithmetic Operators (Iterative Approach - Without + or -)

### 502. Problem:

Given two integers  $a$  and  $b$ , return their sum **without using** the  $+$  or  $-$  operators. This time, solve the problem using an **iterative approach**.

#### Input:

$a = 5, b = 3$

#### Output:

8

#### Solution:

```
function getSum(a, b) {  
    while (b !== 0) {  
        let carry = (a & b) << 1; // Calculate carry  
        a = a ^ b; // Sum without carry  
        b = carry; // Update b with the new carry  
    }  
    return a; // When carry is 0, a contains the final sum  
}  
  
// Example Usage:  
console.log(getSum(5, 3)); // Output: 8  
console.log(getSum(-2, -7)); // Output: -9  
console.log(getSum(10, -3)); // Output: 7  
console.log(getSum(0, 4)); // Output: 4  
console.log(getSum(-5, 5)); // Output: 0
```

## Explanation

### Step 1: Understanding Iterative Bitwise Addition

- **XOR (^)** finds the sum **without carry**.
- **AND (&)** + **Left Shift (<< 1)** finds the carry and moves it to the left.

### Step 2: Iterative Breakdown

Let's take  $a = 5$  (0101 in binary) and  $b = 3$  (0011 in binary).

First Iteration:

```
XOR (5 ^ 3) = 0101 ^ 0011 = 0110 (6 in decimal)
Carry (5 & 3) = 0101 & 0011 = 0001 (1 in decimal)
Shift Carry = 0001 << 1 = 0010 (2 in decimal)
New Values: a = 6, b = 2
```

Second Iteration:

```
XOR (6 ^ 2) = 0110 ^ 0010 = 0100 (4 in decimal)
Carry (6 & 2) = 0110 & 0010 = 0010 (2 in decimal)
Shift Carry = 0010 << 1 = 0100 (4 in decimal)
New Values: a = 4, b = 4
```

Third Iteration:

```
XOR (4 ^ 4) = 0100 ^ 0100 = 0000 (0 in decimal)
Carry (4 & 4) = 0100 & 0100 = 0100 (4 in decimal)
Shift Carry = 0100 << 1 = 1000 (8 in decimal)
New Values: a = 0, b = 8
```

Fourth Iteration:

```
XOR (0 ^ 8) = 0000 ^ 1000 = 1000 (8 in decimal)
Carry (0 & 8) = 0000 & 1000 = 0000 (0 in decimal)
Since b = 0, return a = 8.
```

## Why This Works

- The while loop **keeps computing** sum and carry until carry becomes 0.
- At each iteration, **carry moves left** while sum remains in a.
- When **no carry remains**, a contains the final sum.

## Time & Space Complexity

- **Time Complexity: O(1)** (Limited number of bitwise operations).
- **Space Complexity: O(1)** (Constant space used).

### Number of 1 Bits

#### 503. Problem:

Write a function that takes an unsigned integer and returns the **number of '1' bits** (also known as the **Hamming weight**) it has.

#### Example 1:

##### Input:

```
n = 11; // (Binary representation: 00000000000000000000000000001011)
```

##### Output:

3

#### Solution:

```
function hammingWeight(n) {  
    let count = 0;  
    while (n !== 0) {  
        n = n & (n - 1); // Removes the rightmost 1-bit  
        count++; // Increment count for each removed 1  
    }  
    return count;  
}  
  
// Example Usage:  
console.log(hammingWeight(11)); // Output: 3  
console.log(hammingWeight(128)); // Output: 1  
console.log(hammingWeight(255)); // Output: 8 (Binary: 11111111)  
console.log(hammingWeight(0)); // Output: 0
```

#### Explanation:

##### Step 1: Understanding n & (n - 1) Trick

- This operation **removes the lowest 1 bit** in each iteration.
- The number of times this operation is performed is equal to the number of 1s in n.

## Step 2: Example Walkthrough

Consider  $n = 11$  (000000000000000000000000000000001011 in binary).

### First Iteration:

```
n = 000000000000000000000000000000001011  
n - 1 = 000000000000000000000000000000001010  
n & (n - 1) = 000000000000000000000000000000001010  
count = 1
```

Second Iteration:

```
n = 000000000000000000000000000000001010  
n - 1 = 000000000000000000000000000000001001  
n & (n - 1) = 000000000000000000000000000000001000  
count = 2
```

Third Iteration:

```
n = 000000000000000000000000000000001000  
n - 1 = 000000000000000000000000000000001111  
n & (n - 1) = 000000000000000000000000000000000000  
count = 3
```

Now  $n = 0$ , so we stop. The final count is **3**.

## Time & Space Complexity Analysis

- **Time Complexity:**  $O(k)$ , where  $k$  is the number of 1 bits (maximum 32 for a 32-bit number).
- **Space Complexity:**  $O(1)$ , as we use only a few integer variables.

## Number of 1 Bits (Using Bit Shift Method)

### 504. Problem:

Write a function that takes an unsigned integer and returns the **number of '1' bits** (also known as the **Hamming weight**) using the **bit shifting method**.

#### Input:

```
n = 11; // (Binary representation: 000000000000000000000000000000001011)
```

#### Output:

3

### Solution:

```
function hammingWeightShift(n) {  
    let count = 0;  
    for (let i = 0; i < 32; i++) {  
        // Iterate through all 32 bits  
        if ((n & 1) === 1) count++; // Check if LSB is 1  
        n = n >>> 1; // Unsigned right shift by 1  
    }  
    return count;  
}  
  
// Example Usage:  
console.log(hammingWeightShift(11)); // Output: 3  
console.log(hammingWeightShift(128)); // Output: 1  
console.log(hammingWeightShift(255)); // Output: 8 (Binary: 11111111)  
console.log(hammingWeightShift(0)); // Output: 0
```

### Explanation:

#### Step 1: Checking the Least Significant Bit (LSB)

We use  $(n \& 1) === 1$  to check if the **rightmost bit is 1**.

- $n \& 1$  extracts the **last bit** of  $n$ :
  - If  $n = 11 \rightarrow 000000000000000000000000000000001011$ ,
    - $n \& 1 = 1$  (last bit is 1).

#### Step 2: Right Shifting $n$

We use  $n >>> 1$  to shift  $n$  one bit to the right.

For  $n = 11$  ( $000000000000000000000000000000001011$ ):

1. **First Iteration:**
  - $LSB = 1 \rightarrow Count = 1$
  - Shift Right:  $n = 00000000000000000000000000000000101$
2. **Second Iteration:**
  - $LSB = 1 \rightarrow Count = 2$
  - Shift Right:  $n = 00000000000000000000000000000000010$
3. **Third Iteration:**
  - $LSB = 0 \rightarrow Count remains 2$
  - Shift Right:  $n = 000000000000000000000000000000000001$
4. **Fourth Iteration:**
  - $LSB = 1 \rightarrow Count = 3$
  - Shift Right:  $n = 00$
5. Loop ends, final count = **3**.

### Time & Space Complexity Analysis

- **Time Complexity:**  $O(32) = O(1)$ , since we always loop **32 times**.
- **Space Complexity:**  $O(1)$ , as only a few integer variables are used.

### Number of 1 Bits (Using n & (n - 1) Method)

#### 505. Problem:

Write a function that takes an unsigned integer and returns the **number of '1' bits** (also known as the **Hamming weight**) using the **n & (n - 1) method**.

#### Example 1:

##### Input:

```
n = 11; // (Binary representation: 00000000000000000000000000001011)
```

##### Output:

3

#### Explanation:

- 00000000000000000000000000001011 contains **three** '1' bits.

#### Solution:

```
function hammingWeight(n) {
  let count = 0;
  while (n !== 0) {
    n = n & (n - 1); // Removes the rightmost '1' bit
    count++;
  }
  return count;
}

// Example Usage:
console.log(hammingWeight(11)); // Output: 3
console.log(hammingWeight(128)); // Output: 1
console.log(hammingWeight(255)); // Output: 8 (Binary: 11111111)
console.log(hammingWeight(0)); // Output: 0
```

#### Explanation

##### Step 1: Understanding n & (n - 1)

- $n \& (n - 1)$  removes the rightmost '1' bit.

- Example for  $n = 11$  (000000000000000000000000000000001011):

Iteration	n (Binary)	n - 1 (Binary)	n & (n - 1) Result	Count
1	000000000000000000000000000000001011 000000001011	00000000000000000000000000000000 000000001010	00000000000000000000000000000000 000000001010	1
2	00000000000000000000000000000000 000000001010	00000000000000000000000000000000 000000001001	00000000000000000000000000000000 000000001000	2
3	00000000000000000000000000000000 000000001000	00000000000000000000000000000000 00000000111	00000000000000000000000000000000 000000000000	3

- The loop stops when  $n = 0$ .

### Time & Space Complexity Analysis

- **Time Complexity:**  $O(k)$ , where  $k$  is the number of 1 bits.
- **Space Complexity:**  $O(1)$ , as only a few integer variables are used.

### Number of 1 Bits (Using Bitwise Shift Method)

**506. Problem:** Write a function that takes an unsigned integer and returns the **number of '1'** bits (also known as the **Hamming weight**) using the **bitwise shift method**.

#### Example 1:

##### Input:

```
n = 11; // (Binary representation: 000000000000000000000000000000001011)
```

##### Output:

3

##### Explanation:

- 000000000000000000000000000000001011 contains **three** '1' bits.

##### Solution:

```
function hammingWeight(n) {
    let count = 0;
    while (n !== 0) {
        count += n & 1; // Check the last bit
        n = n >>> 1; // Right shift (unsigned shift)
    }
}
```

```

    return count;
}

// Example Usage:
console.log(hammingWeight(11)); // Output: 3
console.log(hammingWeight(128)); // Output: 1
console.log(hammingWeight(255)); // Output: 8 (Binary: 11111111)
console.log(hammingWeight(0)); // Output: 0

```

### Explanation:

#### Step 1: Understanding Bitwise Shift

- $n \& 1$  checks if the last bit is 1.
- $n >>> 1$  performs **unsigned right shift**, moving bits one position to the right.

**Example for  $n = 11$  (00000000000000000000000000001011)**

Iteration n	n (Binary)	n & 1 (Last Bit)	Count	n >>> 1 (Right Shift)
1	00000000000000000000000000001011	1	1	00000000000000000000000000000101
2	00000000000000000000000000000101	1	2	00000000000000000000000000000010
3	00000000000000000000000000000010	0	2	00000000000000000000000000000001
4	00000000000000000000000000000001	1	3	00000000000000000000000000000000

- The loop stops when  $n = 0$ .

#### Time & Space Complexity Analysis

- **Time Complexity:**  $O(32) = O(1)$ , as we iterate at most 32 times.
- **Space Complexity:**  $O(1)$ , since only a few integer variables are used.

#### Number of 1 Bits (Using Brian Kernighan's Algorithm)

**507. Problem:** Write a function that takes an **unsigned integer** and returns the **number of '1' bits** using **Brian Kernighan's Algorithm**.

### **Example 1:**

## Input:

```
n = 11; // (Binary representation: 0000000000000000000000001011)
```

## Output: 3

## Explanation:

- 000000000000000000000000000000001011 contains **three** '1' bits.

### Solution:

```
function hammingWeight(n) {  
    let count = 0;  
    while (n !== 0) {  
        n = n & (n - 1); // Remove rightmost '1' bit  
        count++;  
    }  
    return count;  
}  
  
// Example Usage:  
console.log(hammingWeight(11)); // Output: 3  
console.log(hammingWeight(128)); // Output: 1  
console.log(hammingWeight(255)); // Output: 8 (Binary: 11111111)  
console.log(hammingWeight(0)); // Output: 0
```

### **Explanation:**

### **Step 1: Understanding n & (n - 1)**

- This operation removes the **rightmost '1' bit** in each iteration.

Iteration n	n (Binary)	n & (n - 1) (Removes Rightmost '1')	Coun t
3	00000000000000000000000000000010 00	00000000000000000000000000000000 00	3

- The loop stops when n = 0.

### Time & Space Complexity Analysis

- Time Complexity:** O(k), where k is the number of '1' bits. In the worst case, k = 32, making it **O(1)**.
- Space Complexity:** O(1), as only a few integer variables are used.

### Counting Bits (Dynamic Programming Approach)

**508. Problem:** Given a non-negative integer num, write a function that returns an array ans of length num + 1, where ans[i] is the number of 1's in the binary representation of i for  $0 \leq i \leq \text{num}$ .

#### Example 1:

##### Input:

num = 5

Output: [0, 1, 1, 2, 1, 2]

##### Explanation:

- Binary representation of 0 = 0 → Number of 1's = 0
- Binary representation of 1 = 1 → Number of 1's = 1
- Binary representation of 2 = 10 → Number of 1's = 1
- Binary representation of 3 = 11 → Number of 1's = 2
- Binary representation of 4 = 100 → Number of 1's = 1
- Binary representation of 5 = 101 → Number of 1's = 2

##### Solution:

```
function countBits(num) {
  let dp = new Array(num + 1).fill(0);
  for (let i = 1; i <= num; i++) {
    dp[i] = dp[i >> 1] + (i & 1); // DP formula
  }
  return dp;
}

// Example Usage:
console.log(countBits(5)); // Output: [0, 1, 1, 2, 1, 2]
```

## **Explanation:**

### **Step-by-Step Breakdown:**

1. **Initialization:**
  - o We initialize an array dp with num + 1 elements, all set to 0. dp[i] will store the number of 1's in the binary representation of i.
2. **Filling the DP Array:**
  - o Start from i = 1 and go till i = num. For each i:
    - dp[i >> 1]: This gives the number of 1's in the binary representation of i / 2.
    - (i & 1): This checks whether i is odd or even. If odd, add 1; if even, add 0.
3. **Result:**
  - o After filling the dp array, we return the array as the final result.

### **Time & Space Complexity Analysis**

- **Time Complexity:** O(n), where n is the input num. We iterate through the array once and each operation is constant time.
- **Space Complexity:** O(n), where n is the input num due to the dp array storing the results.

## **Power of Two**

**509. Problem:** Given an integer n, return true if it is a power of two. Otherwise, return false. An integer n is a power of two if there exists an integer x such that  $n == 2^x$  for some integer x.

### **Example 1:**

#### **Input:**

**N=16**

#### **Output:**

**True**

## **Explanation:**

16 is a power of two since  $16 = 2^4$ .

## **Solution:**

```
function isPowerOfTwo(n) {  
    return n > 0 && (n & (n - 1)) === 0;  
}
```

```
// Example Usage:  
console.log(isPowerOfTwo(16)); // Output: true  
console.log(isPowerOfTwo(3)); // Output: false
```

### Explanation:

### Step-by-Step Breakdown:

1. **Base Case:**
  - o First, check if n is greater than 0. If n is 0 or a negative number, return false as they can't be powers of two.
2. **Power of Two Check:**
  - o Perform the bitwise operation  $n \& (n - 1)$ .
    - For a power of two, this operation results in 0.
    - If this condition holds true, then the number n is a power of two.
3. **Return Result:**
  - o If the condition holds, return true; otherwise, return false.

### Time & Space Complexity Analysis:

- **Time Complexity:** O(1). The operation  $n \& (n - 1)$  is a constant time operation.
- **Space Complexity:** O(1). No additional space is used besides the input.

### Number of 1 Bits

**510. Problem:** Write a function that takes an integer n and returns the number of 1 bits it has (also known as the Hamming weight).

### Example 1:

#### Input:

```
n = 11; // binary representation: 1011
```

#### Output: 3

### Explanation:

The binary representation of 11 is 1011, which has three 1 bits.

### Solution:

```
function hammingWeight(n) {  
    let count = 0;  
    while (n !== 0) {  
        count += n & 1;  
        n >>>= 1; // Use unsigned right shift to handle large numbers
```

```

    }
    return count;
}

// Example Usage:
console.log(hammingWeight(11)); // Output: 3 (binary 1011)
console.log(hammingWeight(128)); // Output: 1 (binary 10000000)

```

### Explanation:

#### Step-by-Step Breakdown:

##### 1. Initialize count:

We start by initializing a count variable to 0, which will store the number of 1 bits.

##### 2. Check LSB:

In the loop, we use the bitwise operation  $n \& 1$  to check the least significant bit. If the result is 1, it means the current bit in the binary representation of n is 1. If it is 1, we increment the count variable by 1.

##### 3. Right Shift:

After checking the least significant bit, we right shift the bits of n using the unsigned right shift ( $>>>$ ) operator. This operation shifts all bits to the right by one place, filling the leftmost bit with 0. This allows us to process the next bit of n.

##### 4. Repeat:

We repeat this process until all bits have been checked (i.e., n becomes 0).

##### 5. Return Count:

Once the loop ends, the count variable will contain the number of 1 bits in the binary representation of n.

### Time & Space Complexity Analysis:

- Time Complexity:**  $O(\log n)$ . The loop runs once for each bit of n, and the number of bits in n is proportional to  $\log n$ .
- Space Complexity:**  $O(1)$ . We are only using a constant amount of space for the count variable and the input number n.

### Number of 1 Bits (Optimized)

**511. Problem:** Write a function that takes an integer n and returns the number of 1 bits in its binary representation, with an optimized approach.

#### Example 1:

##### Input:

```
n = 11; // binary representation: 1011
```

##### Output: 3

### **Explanation:**

The binary representation of 11 is 1011, which contains three 1 bits.

### **Solution:**

```
function hammingWeight(n) {
    let count = 0;
    while (n !== 0) {
        n = n & (n - 1); // Remove the least significant 1 bit
        count++;
    }
    return count;
}

// Example Usage:
console.log(hammingWeight(11)); // Output: 3 (binary 1011)
console.log(hammingWeight(128)); // Output: 1 (binary 10000000)
```

### **Explanation:**

#### **1. Initialization:**

We start by initializing the count variable to 0.

#### **2. Bit Manipulation:**

The key operation here is  $n \& (n - 1)$ . This operation clears the least significant 1 bit of n. For example:

- o If  $n = 1011$  (binary), then  $n - 1 = 1010$ , and  $n \& (n - 1)$  results in 1000.
- o This operation is repeated until all 1 bits are cleared.

#### **3. Loop Execution:**

In each iteration of the loop, the least significant 1 bit is removed, and we increment the count variable to keep track of how many 1 bits have been encountered.

#### **4. Termination:**

Once all the 1 bits have been removed (i.e., n becomes 0), the loop ends, and we return the value of count, which now contains the total number of 1 bits in the binary representation of n.

### **Time & Space Complexity Analysis:**

- **Time Complexity:**  $O(k)$ , where k is the number of 1 bits in the binary representation of n. In the worst case, k could be proportional to the number of bits in n, but it is generally much smaller because most numbers have fewer 1 bits.
- **Space Complexity:**  $O(1)$ . We are using only a constant amount of space for the count variable and the input number n.

### **Counting Bits (Dynamic Programming Approach)**

**512. Problem:** Given a non-negative integer n, for every number i in the range [0, n], calculate the number of 1 bits in its binary representation and return them as an array.

### **Example 1:**

**Input:****N=5****Output:**

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

**Explanation:**

The binary representations of the numbers in the range [0, 5] are:

- 0 → 0 → 0 1 bits
- 1 → 1 → 1 1 bits
- 2 → 10 → 1 1 bits
- 3 → 11 → 2 1 bits
- 4 → 100 → 1 1 bits
- 5 → 101 → 2 1 bits

**Solution:**

```
function countBits(n) {  
    let dp = new Array(n + 1); // Array to store the count of 1 bits for each number  
    dp[0] = 0; // Base case: 0 has 0 ones  
  
    // Iterate over each number from 1 to n  
    for (let i = 1; i <= n; i++) {  
        // Use the relation dp[i] = dp[i >> 1] + (i & 1)  
        dp[i] = dp[i >> 1] + (i & 1);  
    }  
  
    return dp; // Return the result array  
}  
  
// Example Usage:  
console.log(countBits(5)); // Output: [0, 1, 1, 2, 1, 2]
```

**Explanation:****1. Base Case:**

We initialize  $dp[0]$  to 0 since the binary representation of 0 contains 0 1 bits.

**2. Loop through Numbers:**

Starting from  $i = 1$ , for each number, we compute the number of 1 bits using the relation:

- $dp[i] = dp[i >> 1] + (i \& 1)$ 
  - $i >> 1$  is the result of right-shifting  $i$  by one bit, essentially dividing  $i$  by 2.
  - $i \& 1$  checks if  $i$  is odd (1) or even (0), since odd numbers have a 1 bit in their least significant position.

### 3. Fill the Array:

We continue this process for all numbers up to n. As we are using previously computed values ( $dp[i >> 1]$ ), this leads to efficient computation.

### 4. Return the Result:

After processing all numbers up to n, the dp array will contain the number of 1 bits for all numbers in the range [0, n].

## Time & Space Complexity Analysis:

- **Time Complexity:**  $O(n)$ . We are iterating through all numbers from 1 to n and performing constant-time operations (bit-shifting and bitwise AND) for each number.
- **Space Complexity:**  $O(n)$ . We are storing the results for all numbers from 0 to n in the dp array.

## Count Bits - Improved Space Complexity

**513. Problem:** Given a non-negative integer n, for every number i in the range [0, n], calculate the number of 1 bits in its binary representation and return them as an array. The problem asks for an optimized solution with better space complexity compared to the previous approach.

### Example 1:

#### Input:

N=5

#### Output:

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

#### Explanation:

The binary representations of the numbers in the range [0, 5] are:

- $0 \rightarrow 0 \rightarrow 0$  1 bits
- $1 \rightarrow 1 \rightarrow 1$  1 bits
- $2 \rightarrow 10 \rightarrow 1$  1 bits
- $3 \rightarrow 11 \rightarrow 2$  1 bits
- $4 \rightarrow 100 \rightarrow 1$  1 bits
- $5 \rightarrow 101 \rightarrow 2$  1 bits

#### Solution:

```
function countBits(n) {  
    let dp = new Array(n + 1); // Array to store the count of 1 bits for each number  
    dp[0] = 0; // Base case: 0 has 0 ones  
  
    // Iterate over each number from 1 to n  
    for (let i = 1; i <= n; i++) {  
        // Use the relation dp[i] = dp[i >> 1] + (i & 1)  
        dp[i] = dp[i >> 1] + (i & 1);  
    }  
}
```

```

    }

    return dp; // Return the result array
}

// Example Usage:
console.log(countBits(5)); // Output: [0, 1, 1, 2, 1, 2]

```

### **Explanation:**

#### **1. Base Case:**

Initialize  $dp[0]$  to 0 since 0 has no 1 bits in its binary representation.

#### **2. Loop through Numbers:**

From  $i = 1$  to  $n$ , for each  $i$ , calculate the number of 1 bits using the formula:

- o  $dp[i] = dp[i >> 1] + (i \& 1)$ 
  - $i >> 1$  is the result of right-shifting  $i$  by one bit (essentially  $i / 2$ ).
  - $i \& 1$  checks if  $i$  is odd (1) or even (0).

#### **3. Memory Efficiency:**

This solution only requires an array of size  $n + 1$ , and the array is filled in a way that reuses the previous computations, making the space complexity  $O(n)$ .

#### **4. Return the Result:**

After the loop completes, return the array  $dp$  containing the number of 1 bits for each number from 0 to  $n$ .

### **Time Complexity:**

- **Time Complexity:**  $O(n)$  — The loop runs for  $n$  numbers, and for each number, the computation takes constant time.
- **Space Complexity:**  $O(n)$  — We are using an array of size  $n + 1$  to store the results.

### **Missing Number (Array 1 to n)**

**514. Problem:** Given an array  $nums$  containing  $n$  distinct numbers taken from the range  $[1, n + 1]$ , find the one number that is missing from the array.

### **Example 1:**

#### **Input:**

```
nums = [3, 7, 1, 2, 8, 4, 5];
```

#### **Output: 6**

**Explanation:** The array contains the numbers from the range  $[1, 8]$  except for 6, so the missing number is 6.

### **Solution:**

```
function missingNumber(nums) {
```

```

let n = nums.length;

// Sum of numbers from 1 to n + 1
let totalSum = ((n + 1) * (n + 2)) / 2;

// Sum of numbers in the array
let arraySum = nums.reduce((sum, num) => sum + num, 0);

// The missing number is the difference between the total sum and array sum
return totalSum - arraySum;
}

// Example Usage:
console.log(missingNumber([3, 7, 1, 2, 8, 4, 5])); // Output: 6

```

### **Explanation:**

- 1. Sum Formula for First n + 1 Numbers:**

We use the formula to calculate the sum of numbers from 1 to n + 1. This gives us the sum we would expect if all numbers from 1 to n + 1 were present in the array.

$$\text{Sum} = \frac{(n+1) \times (n+2)}{2}$$

- 2. Sum of Array:**

We calculate the sum of all the numbers present in the array. This can be done using the reduce method in JavaScript to sum up all the numbers.

- 3. Finding the Missing Number:**

The missing number is simply the difference between the expected total sum (totalSum) and the sum of the numbers in the array (arraySum). The difference gives us the missing number.

### **Time Complexity:**

- Time Complexity:** O(n) — We loop through the array once to calculate the sum of the numbers in it.
- Space Complexity:** O(1) — We are using only a constant amount of extra space (besides the input array).

### **Find the Missing Number in a Sequence**

**515. Problem:** You are given an array of integers from 1 to n+1 but with exactly one number missing. The array is in no particular order. Your task is to find the missing number.

### **Example 1:**

#### **Input:**

```
nums = [1, 2, 3, 5];
```

## Output: 4

**Explanation:** The array contains the numbers from the range 1 to 5 except for 4, so the missing number is 4.

## Solution:

```
function missingNumber(nums) {  
    let n = nums.length;  
    let xorResult = n; // Start with n as it will be XOR'd with the range numbers  
  
    // XOR all the numbers in the array  
    for (let i = 0; i < n; i++) {  
        xorResult ^= nums[i];  
    }  
  
    // XOR with all numbers from 0 to n  
    for (let i = 0; i < n; i++) {  
        xorResult ^= i;  
    }  
  
    return xorResult; // The missing number  
}  
  
// Example Usage:  
console.log(missingNumber([1, 2, 3, 5])); // Output: 4
```

## Explanation:

### 1. XOR All Numbers in the Array:

The XOR of all elements in the given array is calculated. By XORing all the numbers together, we eliminate the numbers that exist in both the array and the range [1, n+1].

### 2. XOR with the Range:

We then XOR the result with all the numbers in the range [0, n]. This ensures that every number from 1 to n cancels out, and only the missing number remains in the result.

### 3. Final Result:

The final result is the number that did not appear in the array and is missing from the range.

## Time Complexity:

- **Time Complexity:**  $O(n)$  — We loop through the array and the range [1, n+1] once to perform XOR operations.
- **Space Complexity:**  $O(1)$  — We use only a constant amount of space.

## Find the Missing Number (Alternative Approach)

**516. Problem:** You are given an array of integers from 0 to n. However, exactly one number from the array is missing. Your task is to find the missing number.

**Example 1:**

**Input:**

```
nums = [3, 7, 1, 2, 8, 4, 5];
```

**Output: 6**

**Solution:**

```
function missingNumber(nums) {  
    let n = nums.length; // Array length gives us the largest number `n`  
    let expectedSum = (n * (n + 1)) / 2; // Sum of numbers from 0 to n  
    let actualSum = nums.reduce((sum, num) => sum + num, 0); // Sum of numbers in the array  
  
    return expectedSum - actualSum; // The missing number  
}  
  
// Example Usage:  
console.log(missingNumber([3, 7, 1, 2, 8, 4, 5])); // Output: 6
```

**Explanation:**

**1. Calculate Expected Sum:**

We calculate the expected sum of all numbers from 0 to n using the formula:

$$\text{Expected Sum} = \frac{n(n+1)}{2}$$

This sum represents the total if there were no missing numbers in the array.

**2. Calculate Actual Sum:**

Using `reduce()`, we sum all the numbers present in the array.

**3. Find Missing Number:**

The missing number can be found by subtracting the actual sum from the expected sum. This difference gives us the number that is missing from the array.

**Time Complexity:**

- **Time Complexity:** O(n) — We traverse the array once to calculate the sum.
- **Space Complexity:** O(1) — No additional space is used.

### Find the Missing Number in a Sequence with Duplicates

**517. Problem:** Given a sequence of integers where one number from the range 1 to n is missing, and there are duplicates in the sequence, find the missing number.

### Example 1:

#### Input:

```
nums = [1, 3, 4, 2, 2];
```

#### Output: 5

**Explanation:** The array contains the numbers 1, 2, 2, 3, 4, but the number 5 is missing.

#### Solution:

```
function findMissingNumber(nums) {  
    let n = nums.length;  
    let expectedSum = (n * (n + 1)) / 2;  
    let actualSum = 0;  
    let seen = new Set();  
  
    // Sum the unique numbers and store them in the set  
    for (let num of nums) {  
        if (!seen.has(num)) {  
            actualSum += num;  
            seen.add(num);  
        }  
    }  
  
    // The missing number is the difference between expected and actual sums  
    return expectedSum - actualSum;  
}  
  
// Example Usage:  
console.log(findMissingNumber([1, 3, 4, 2, 2])); // Output: 5
```

#### Explanation:

##### 1. Calculate Expected Sum:

The expected sum is the sum of the integers from 1 to n using the formula:

$$\text{Expected Sum} = \frac{n(n+1)}{2}$$

##### 2. Track Unique Numbers:

A **Set** is used to store unique numbers from the sequence. As we iterate over the array, if a number is not in the set, we add it to the sum and the set. This ensures we only count each unique number once.

### 3. Find Missing Number:

The missing number is the difference between the expected sum and the sum of unique numbers found in the array.

#### Time Complexity:

- **Time Complexity:**  $O(n)$  — We iterate through the array once, adding unique numbers to the set.
- **Space Complexity:**  $O(n)$  — Space for the set to store unique numbers.

#### Find the Missing Number from Array

**518. Problem Statement:** Given an array of  $n - 1$  integers, which is a permutation of the integers from 1 to  $n$  with one number missing, find the missing number.

#### Example 1:

##### Input:

```
nums = [3, 7, 1, 2, 8, 4, 5];
```

##### Output:

6

**Explanation:** The array contains the numbers from 1 to 8 except 6.

#### Solution:

```
function findMissingNumber(nums) {  
    let n = nums.length + 1; // Since array has n-1 elements, the total length is n  
    let expectedSum = (n * (n + 1)) / 2;  
    let actualSum = nums.reduce((sum, num) => sum + num, 0);  
  
    return expectedSum - actualSum;  
}  
  
// Example Usage:  
console.log(findMissingNumber([3, 7, 1, 2, 8, 4, 5])); // Output: 6
```

#### Explanation:

##### 1. Calculate Expected Sum:

The expected sum is the sum of integers from 1 to  $n$ :

$$\text{Total Sum} = n \times (n+1) / 2$$

$$\text{Total Sum} = \frac{n(n+1)}{2}$$

$$\text{Total Sum} = 2n(n+1)$$

## 2. Calculate Actual Sum:

We calculate the sum of the given array using the **reduce** method. This gives us the sum of the  $n - 1$  elements present in the array.

## 3. Find Missing Number:

The missing number is simply the difference between the expected sum and the actual sum:

$$\text{Missing Number} = \text{Total Sum} - \text{Actual Sum}$$

$$\text{Missing Number} = \frac{n(n+1)}{2} - \text{Actual Sum}$$

## Time Complexity:

- Time Complexity:**  $O(n)$  — We compute the sum of the array using the reduce method, which iterates through the array once.
- Space Complexity:**  $O(1)$  — We use a constant amount of extra space.

## Reversing Binary Patterns

### 519. Problem:

Given a 32-bit unsigned integer, reverse the bits of it and return the result.

For example, if the input is 43261596 (in binary 0000001010010100000111010011100), the output should be 964176192 (in binary 00111001011110000010100101000000).

#### Example 1:

##### Input:

```
n = 43261596;
```

##### Output:

964176192

**Explanation:** The input is in binary 0000001010010100000111010011100. After reversing, we get the binary value 00111001011110000010100101000000, which is the decimal number 964176192.

#### Solution:

```
var reverseBits = function (n) {
    let result = 0;
    for (let i = 0; i < 32; i++) {
        // Get the rightmost bit of n
        result = (result << 1) | (n & 1);
        // Shift n to the right to process the next bit
        n >>>= 1;
    }
    return result;
}
```

```

    }
    return result;
};

// Example usage:
console.log(reverseBits(43261596)); // Output: 964176192

```

### **Explanation:**

1. **Initialize the Result Variable:**

We start with a variable result set to 0, which will store the reversed bits.

2. **Iterate through each bit (32 iterations):**

In each iteration, we:

- o Extract the rightmost bit of n using  $n \& 1$ .
- o Left shift result by 1 to make room for the next bit.
- o Add the extracted bit to the result using the bitwise OR ( $\|$ ).
- o Right shift n by 1 to process the next bit.

3. **Finish the Loop:**

Once the loop completes 32 iterations, the result will contain the reversed bits.

### **Time Complexity:**

- **Time Complexity:**  $O(1)$  — The number of iterations is constant (32) because we are processing a 32-bit number.
- **Space Complexity:**  $O(1)$  — We use a constant amount of extra space to store the result.

### **Reversing Binary Patterns**

#### **520. Problem: Given a 32-bit unsigned integer n, reverse its bits and return the reversed number.**

##### **Example 1:**

##### **Input:**

$n = 43261596$

##### **Output:**

964176192

**Explanation:** The binary representation of 43261596 is 0000001010010100001111010011100. It becomes 0011100101110000010100101000000 after reversing the bits, which is 964176192.

##### **Solution:**

```

var reverseBits = function (n) {
  let result = 0;

```

```

for (let i = 0; i < 32; i++) {
    result = (result << 1) | (n & 1); // Shift left and add the last bit of n
    n >>= 1; // Shift n right by 1 to move to the next bit
}
return result >>> 0; // Convert the result to an unsigned 32-bit integer
};

// Example usage:
console.log(reverseBits(43261596)); // Output: 964176192

```

### **Explanation:**

**1. Initialize result:**

Start with result = 0, which will hold the reversed bits.

**2. Extract and Reverse Each Bit:**

In the loop:

- n & 1 extracts the last bit of n.
- result = (result << 1) | (n & 1) shifts the result left by one position and adds the extracted bit (either 0 or 1).
- n >>= 1 shifts n right by one position to prepare for the next bit extraction.

**3. 32 Iterations:**

Since n is a 32-bit integer, we repeat the above process 32 times to reverse all the bits.

**4. Return the Result:**

After the loop, result holds the reversed bits. We return result >>> 0 to ensure the result is treated as an unsigned 32-bit integer (in case negative values appear due to JavaScript's handling of bitwise operations).

### **Time Complexity:**

- **Time Complexity:** O(1) — The algorithm always runs in 32 iterations, making it constant time for a 32-bit integer.
- **Space Complexity:** O(1) — We use only a constant amount of space for the result variable.

### **Count the Number of 1 Bits**

**521. Problem:** Write a function that takes a number and returns the number of 1 bits it has. You need to implement the function to count the number of 1 bits in the binary representation of a given number.

### **Example 1:**

#### **Input:**

**N=11**

#### **Output:**

### 3

**Explanation:** The binary representation of 11 is 1011, which contains three 1 bits.

**Solution:**

```
var hammingWeight = function (n) {  
    let count = 0;  
    while (n !== 0) {  
        count += n & 1; // Check if the least significant bit is 1  
        n >>>= 1; // Right shift n to move to the next bit  
    }  
    return count;  
};  
  
// Example usage:  
console.log(hammingWeight(11)); // Output: 3
```

**Explanation:**

**1. Initialize Count:**

We start by initializing count = 0, which will hold the number of 1 bits in n.

**2. Check the Least Significant Bit:**

The condition n & 1 checks whether the least significant bit of n is 1. If it is, we increment count.

**3. Right Shift n:**

After checking the least significant bit, we shift n right by 1 using n >>>= 1 to check the next bit in the following iteration.

**4. Repeat Until n Becomes 0:**

This process continues until n becomes 0. At that point, all the bits have been checked, and count holds the total number of 1 bits.

**5. Return the Count:**

Finally, we return the count, which gives us the number of 1 bits in the binary representation of n.

**Time Complexity:**

- **Time Complexity:** O(k) — Where k is the number of bits in the integer n. For typical 32-bit or 64-bit integers, this is constant time (O(1)).
- **Space Complexity:** O(1) — We only use a constant amount of extra space for the count variable.

## Reversing Binary Patterns

**522. Problem:** Given a 32-bit signed integer, reverse the bits of the number and return it as a 32-bit unsigned integer.

### Example 1:

#### Input:

```
n = 43261596;
```

#### Output:

964176192

**Explanation:** The binary representation of 43261596 is 00000010100101000001111010011100. Reversing this gives 0011100101110000010100101000000, which is 964176192 in decimal.

#### Solution:

```
var reverseBits = function (n) {
    let reversed = 0;
    for (let i = 0; i < 32; i++) {
        reversed <<= 1; // Shift the result left by 1 to make space for the next bit
        reversed |= n & 1; // Add the least significant bit of n to reversed
        n >>>= 1; // Right shift n to move to the next bit
    }
    return reversed >>> 0; // Ensure we return an unsigned 32-bit integer
};

// Example usage:
console.log(reverseBits(43261596)); // Output: 964176192
```

#### Explanation:

1. **Initialization:**  
We start by initializing reversed = 0, which will hold the result of the reversed bits.
2. **Loop Through 32 Bits:**  
The loop runs for 32 iterations, since we are dealing with a 32-bit number.
3. **Left Shift the Result:**  
In each iteration, we shift reversed left by 1 bit (reversed <<= 1) to make space for the new bit.
4. **Extract the Least Significant Bit:**  
The expression n & 1 extracts the least significant bit of n. We then add it to reversed using the bitwise OR operation (reversed |= (n & 1)).
5. **Right Shift n:**  
After adding the least significant bit to reversed, we right shift n by 1 bit (n >>>= 1) to process the next bit.
6. **Ensure 32-bit Unsigned Integer:**  
Finally, after the loop finishes, we return reversed >>> 0. This ensures that the result is treated as an unsigned 32-bit integer, effectively removing any sign bits.

## Time Complexity:

- **Time Complexity:** O(1) — The number of bits is fixed at 32, so the time complexity is constant.
- **Space Complexity:** O(1) — We only use a fixed amount of extra space (for the reversed variable).

## Reversing Binary Patterns

**523. Problem: Given a 32-bit unsigned integer, reverse its bits and return the resulting number.**

### Example 1:

```
Input: n = 43261596
Output: 964176192
Explanation: The binary representation of 43261596 is
00000010100101000001111010011100, and its reverse is
0011100101110000010100101000000, which is 964176192.
```

### Solution:

```
var reverseBits = function (n) {
    let reversed = 0;
    for (let i = 0; i < 32; i++) {
        reversed = (reversed << 1) | (n & 1);
        n >>= 1;
    }
    return reversed >>> 0; // Unsigned right shift to ensure the result is treated as unsigned.
};
```

### Explanation:

#### 1. Understanding the Problem:

- We are given a 32-bit unsigned integer, which means the number will have exactly 32 bits.
- The task is to reverse the order of the bits in the binary representation of the number.

#### 2. Approach:

- We'll use a bit manipulation technique to reverse the bits.
- We initialize the result to 0 and then iterate through the 32 bits of the number:
  - For each bit in the input number, shift the result left by 1 and add the current bit from the number (using bitwise AND).
  - Then, shift the input number right to process the next bit.
- After processing all 32 bits, the result will contain the reversed bits.

#### 3. Steps:

- Initialize a variable reversed to store the reversed number, starting with 0.
- Loop through each bit of the input number:
  - Shift the reversed number left by 1 to make room for the next bit.

- Use the bitwise AND operation to extract the last bit of the input number.
  - Add this extracted bit to reversed.
  - Shift the input number right by 1 to discard the last bit.
- After 32 iterations, return the reversed number.

#### 4. Why it Works:

- By shifting the reversed number left, we make space to add the next bit, and by shifting the input number right, we discard the processed bit. This effectively builds the reversed bit pattern from left to right.

**Time Complexity:** O(1) — Constant time because we always process exactly 32 bits.

**Space Complexity:** O(1) — Constant space for storing the reversed number.

## 7. Interval & Range-Based Problems

### Efficient Range Insertions in an Array

#### 524. Problem:

Given a list of non-overlapping intervals intervals where intervals[i] = [start\_i, end\_i], insert a new interval newInterval = [start, end] into intervals such that the intervals remain sorted in ascending order by their start times. You need to return the **new list of intervals** after the insertion.

#### Example 1:

Input: intervals = [[1,3],[6,9]], newInterval = [2,5]

Output: [[1,5],[6,9]]

Explanation: The new interval [2,5] overlaps with [1,3], so we merge them into [1,5].

#### Solution:

```
var insert = function (intervals, newInterval) {
  let result = [];
  let i = 0;

  // Add all intervals that come before newInterval
  while (i < intervals.length && intervals[i][1] < newInterval[0]) {
    result.push(intervals[i]);
    i++;
  }

  // Merge intervals that overlap with newInterval
  while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
    newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
    newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
    i++;
  }

  result.push(newInterval);
  return result;
}
```

```

    }
    result.push(newInterval); // Add the merged interval

    // Add remaining intervals after newInterval
    while (i < intervals.length) {
        result.push(intervals[i]);
        i++;
    }

    return result;
};

```

### **Explanation:**

1. **Initialize an empty result array** to store the final list of intervals.
2. **Iterate through the intervals** and find intervals that are:
  - o Completely before the new interval (add them to the result).
  - o Overlapping with the new interval (merge them).
  - o Completely after the new interval (add the merged new interval and then the remaining intervals).
3. **Edge Case:** If the list is empty, we simply return the new interval as the only element in the result.

### **Why it Works:**

- The approach efficiently merges overlapping intervals and keeps the list sorted. By processing intervals in a single pass, it avoids unnecessary comparisons or backtracking.

### **Time Complexity:**

- $O(n)$ , where  $n$  is the number of intervals. We go through the list of intervals once, comparing and inserting intervals.

### **Space Complexity:**

- $O(n)$ , where  $n$  is the number of intervals. We store the resulting list of intervals, which can be as large as the original list plus one new interval.

## **Efficient Range Insertions in an Array**

### **525. Problem:**

Given a list of non-overlapping intervals intervals where  $\text{intervals}[i] = [\text{start}_i, \text{end}_i]$ , insert a new interval  $\text{newInterval} = [\text{start}, \text{end}]$  into intervals such that the intervals remain sorted in ascending order by their start times. You need to return the **new list of intervals** after the insertion.

### Example 1:

Input: intervals = [[1,3],[6,9]], newInterval = [2,5]

Output: [[1,5],[6,9]]

Explanation: The new interval [2,5] overlaps with [1,3], so we merge them into [1,5].

### Solution:

```
var insert = function (intervals, newInterval) {  
    let result = [];  
    let i = 0;  
  
    // Add all intervals that come before newInterval  
    while (i < intervals.length && intervals[i][1] < newInterval[0]) {  
        result.push(intervals[i]);  
        i++;  
    }  
  
    // Merge intervals that overlap with newInterval  
    while (i < intervals.length && intervals[i][0] <= newInterval[1]) {  
        newInterval[0] = Math.min(newInterval[0], intervals[i][0]);  
        newInterval[1] = Math.max(newInterval[1], intervals[i][1]);  
        i++;  
    }  
    result.push(newInterval); // Add the merged interval  
  
    // Add remaining intervals after newInterval  
    while (i < intervals.length) {  
        result.push(intervals[i]);  
        i++;  
    }  
  
    return result;  
};
```

### Explanation:

1. **Initialize an empty result array** to store the final list of intervals.
2. **Iterate through the intervals** and find intervals that are:
  - o Completely before the new interval (add them to the result).
  - o Overlapping with the new interval (merge them).
  - o Completely after the new interval (add the new interval and the remaining intervals).
3. **Handle the edge cases** where the list is empty or where the new interval is inserted at the beginning or end.

## Why it Works:

- The approach efficiently merges overlapping intervals and keeps the list sorted. By processing intervals in a single pass, it avoids unnecessary comparisons or backtracking.

## Time Complexity:

- $O(n)$ , where  $n$  is the number of intervals. We go through the list of intervals once, comparing and inserting intervals.

## Space Complexity:

- $O(n)$ , where  $n$  is the number of intervals. We store the resulting list of intervals, which can be as large as the original list plus one new interval.

## Remove Interval

**526. Problem:** Given a list of intervals intervals where each interval is represented by a pair of integers [start, end] (inclusive), remove all intervals that overlap with a given interval toRemove. You need to return the list of intervals after removal, sorted by the start time.

### Example 1:

Input: intervals = [[0,2],[3,4],[5,7]], toRemove = [1,6]

Output: [[0,1],[6,7]]

Explanation: Intervals [3,4] and [5,7] are removed because they overlap with [1,6].

### Solution:

```
var removeInterval = function (intervals, toRemove) {
    let result = [];
    let [startRemove, endRemove] = toRemove;

    for (let [start, end] of intervals) {
        // If the current interval ends before the toRemove interval starts, no overlap, add to result
        if (end < startRemove) {
            result.push([start, end]);
        }
        // If the current interval starts after the toRemove interval ends, no overlap, add to result
        else if (start > endRemove) {
            result.push([start, end]);
        }
        // If there is an overlap
        else {
```

```

// If the current interval starts before toRemove, add the part before toRemove
if (start < startRemove) {
    result.push([start, startRemove]);
}
// If the current interval ends after toRemove, add the part after toRemove
if (end > endRemove) {
    result.push([endRemove, end]);
}
}

return result;
};

```

### **Explanation:**

#### **1. Iterate through the intervals:**

- For each interval [start, end]:
  - If it ends before the toRemove interval starts ( $end < startRemove$ ), add the interval as is to result.
  - If it starts after the toRemove interval ends ( $start > endRemove$ ), add the interval as is to result.
  - If there's an overlap, split the interval:
    - If the current interval starts before toRemove, push  $[start, startRemove]$  (before the removal).
    - If the current interval ends after toRemove, push  $[endRemove, end]$  (after the removal).

#### **2. Why it works:**

- The solution processes each interval once and decides whether to add, remove, or split it based on the overlap with toRemove.

#### **3. Edge Cases Handled:**

- Full overlap: A single interval becomes multiple intervals after the removal.
- No overlap: The interval is left as is.
- The list might have intervals that are either before or after the toRemove interval, which are simply added without modification.

### **Why it Works:**

- The problem is efficiently solved by processing intervals only once, ensuring that the solution runs in linear time with respect to the number of intervals. Since the input intervals are already sorted, no extra sorting is required.

### **Time Complexity:**

- $O(n)$ , where  $n$  is the number of intervals. We only iterate over the list of intervals once.

## Space Complexity:

- $O(n)$ , where  $n$  is the number of intervals. We store the resulting list of intervals.

## Merge Intervals

**527. Problem:** Given a collection of intervals, merge all overlapping intervals. You need to return the merged intervals sorted by their starting point.

### Example 1:

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Intervals [1,3] and [2,6] overlap, so they are merged into [1,6].

## Solution:

```
var merge = function (intervals) {
    if (intervals.length === 0) return [];

    // Step 1: Sort intervals by the start of each interval
    intervals.sort((a, b) => a[0] - b[0]);

    let result = [intervals[0]];

    for (let i = 1; i < intervals.length; i++) {
        let lastInterval = result[result.length - 1];
        let currentInterval = intervals[i];

        // Step 2: Check if intervals overlap
        if (lastInterval[1] >= currentInterval[0]) {
            // Merge intervals
            lastInterval[1] = Math.max(lastInterval[1], currentInterval[1]);
        } else {
            // No overlap, add the current interval to the result
            result.push(currentInterval);
        }
    }

    return result;
};
```

## Explanation:

### 1. Sorting the intervals:

- First, we sort the intervals by their starting point. This helps us easily check whether two consecutive intervals overlap.

**2. Iterating through the intervals:**

- We start with the first interval in result.
- For each subsequent interval, we check if it overlaps with the last interval in result (the one at the end of the list).
- If the current interval overlaps with the last one, we merge them by adjusting the end of the last interval to be the maximum of the two ends.
- If they do not overlap, we simply add the current interval to the result list.

**3. Return the merged intervals:**

- After processing all intervals, the result will contain the merged non-overlapping intervals.

**4. Edge Cases Handled:**

- If the input is an empty list, we simply return an empty list.
- If no intervals overlap, the original intervals remain unchanged in the result.

**Time Complexity:**

- Sorting the intervals takes  $O(n \log n)$ , and iterating through the intervals takes  $O(n)$ , where  $n$  is the number of intervals. Hence, the overall time complexity is  $O(n \log n)$ .

**Space Complexity:**

- We store the merged intervals in an additional result list, so the space complexity is  $O(n)$ .

**Efficient Range Insertions in an Array**

**528. Problem:** You are given a list of intervals where each interval consists of two integers representing the start and end of an interval. You are also given a new interval that needs to be inserted into the list of intervals. Your task is to insert the new interval into the correct position and merge any overlapping intervals.

**Return the resulting list of intervals after insertion and merging.**

**Example 1:**

Input: intervals = [[1,3],[6,9]], newInterval = [2,5]

Output: [[1,5],[6,9]]

Explanation: The new interval [2,5] overlaps with the first interval [1,3], so we merge them into [1,5].

**Solution:**

```
var insert = function (intervals, newInterval) {
  let result = [];
  let i = 0;

  // Step 1: Add all intervals ending before newInterval starts
```

```

while (i < intervals.length && intervals[i][1] < newInterval[0]) {
    result.push(intervals[i]);
    i++;
}

// Step 2: Merge overlapping intervals
while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
    newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
    newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
    i++;
}

result.push(newInterval); // Add the merged interval

// Step 3: Add all intervals starting after newInterval ends
while (i < intervals.length) {
    result.push(intervals[i]);
    i++;
}

return result;
};

```

### Explanation:

1. **Initial Setup:**
  - o We initialize an empty result array that will hold the merged intervals.
  - o We start iterating through the existing intervals.
2. **Step 1 - Add non-overlapping intervals:**
  - o We first add intervals that end before the new interval starts. These intervals don't need any modification and can be added directly to the result.
3. **Step 2 - Merge overlapping intervals:**
  - o If the current interval overlaps with the new interval, we merge them. The merging is done by updating the start and end values of the new interval to the minimum start and maximum end of the two overlapping intervals.
4. **Step 3 - Add remaining intervals:**
  - o After merging, we add any remaining intervals that start after the new interval ends to the result.
5. **Return the result:**
  - o Finally, we return the result array which contains the correctly merged and inserted intervals.

### Time Complexity:

- Sorting the intervals is not required as we assume they are already sorted.
- Iterating through the list of intervals is O(n), where n is the number of intervals.
- Overall, the time complexity is O(n).

## Space Complexity:

- We are using an additional result array to store the merged intervals, so the space complexity is O(n).

## Merge Intervals

**529. Problem:** Given an array of intervals where  $\text{intervals}[i] = [\text{start}_i, \text{end}_i]$ , merge all overlapping intervals and return an array of the non-overlapping intervals that cover all the intervals in the input.

### Example 1:

Input:  $\text{intervals} = [[1,3],[2,6],[8,10],[15,18]]$

Output:  $[[1,6],[8,10],[15,18]]$

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

## Solution:

```
var merge = function (intervals) {
    if (intervals.length === 0) return [];

    // Step 1: Sort intervals by their start times
    intervals.sort((a, b) => a[0] - b[0]);

    let merged = [];

    for (let interval of intervals) {
        // If merged is empty or no overlap, just add the current interval
        if (merged.length === 0 || merged[merged.length - 1][1] < interval[0]) {
            merged.push(interval);
        } else {
            // There is an overlap, merge the intervals
            merged[merged.length - 1][1] = Math.max(
                merged[merged.length - 1][1],
                interval[1]
            );
        }
    }

    return merged;
};
```

## Explanation:

To merge intervals, the general approach is:

1. **Sort** the intervals by their start times.
2. **Iterate** through the sorted intervals and merge overlapping intervals as needed.

### Steps:

1. **Sort the intervals** based on their start time.
2. Initialize an empty array merged to hold the merged intervals.
3. For each interval in the sorted list:
  - o If the merged array is empty or the current interval doesn't overlap with the last interval in merged, add it to merged.
  - o If there is an overlap (i.e., the current interval's start time is less than or equal to the end time of the last interval in merged), merge them by updating the end time of the last interval in merged.
4. Return the merged array.

### Why it Works:

- Sorting the intervals ensures that we always have intervals in increasing order of their start times, so we can easily identify overlapping intervals.
- By iterating through the intervals and merging them when necessary, we can create a final list of non-overlapping intervals.

### Time Complexity:

- Sorting the intervals takes  $O(n \log n)$ , where  $n$  is the number of intervals.
- Merging the intervals takes  $O(n)$ , as we iterate through the intervals once.
- Overall time complexity:  $O(n \log n)$ .

### Space Complexity:

- The space complexity is  $O(n)$ , as we are storing the merged intervals in the merged array.

## Efficient Range Insertions in an Array

### 530. Problem:

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start time.

### Example 1:

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
Explanation: After inserting [2,5], the intervals become [1,5],[6,9].
```

### Solution:

```
var insert = function (intervals, newInterval) {
    let result = [];
    let i = 0;

    // Step 1: Add intervals that come before the new interval
    while (i < intervals.length && intervals[i][1] < newInterval[0]) {
        result.push(intervals[i]);
        i++;
    }

    // Step 2: Merge the new interval with overlapping intervals
    while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
        newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
        newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
        i++;
    }
    result.push(newInterval);

    // Step 3: Add the intervals that come after the new interval
    while (i < intervals.length) {
        result.push(intervals[i]);
        i++;
    }

    return result;
};
```

### Explanation:

1. **Step 1: Add intervals before the new interval:**
  - o We first iterate through the given intervals and add any intervals that end before the new interval starts. This is because these intervals will not overlap with the new interval, and we can safely add them to the result.
2. **Step 2: Merge overlapping intervals:**
  - o After identifying the intervals that do not overlap, we begin merging the overlapping intervals. We update the start of the new interval to be the minimum of the start of the new interval and the current interval's start. We also update the end of the new interval to be the maximum of the current interval's end and the new interval's end.
3. **Step 3: Add intervals after the new interval:**
  - o Finally, we add all the remaining intervals that start after the new interval ends, as they will not overlap with the new interval.
4. **Return Result:**
  - o Once all the intervals have been processed, the result array contains the merged intervals, which is returned.

## Example Walkthrough:

For the input  $[[1,3],[6,9]]$  and new interval  $[2,5]$ :

1. **Step 1: Add intervals before the new interval:**
  - o There are no intervals that end before  $[2,5]$ , so we move to the next step.
2. **Step 2: Merge overlapping intervals:**
  - o The first interval  $[1,3]$  overlaps with  $[2,5]$ , so we update the start time to  $\min(1,2) = 1$  and the end time to  $\max(3,5) = 5$ . Thus, we have  $[1,5]$ .
3. **Step 3: Add intervals after the new interval:**
  - o The next interval  $[6,9]$  starts after  $[1,5]$ , so it is added to result.
4. **Return Result:**
  - o The final merged intervals are  $[[1,5],[6,9]]$ .

## Merge Intervals

**531. Problem: Given a collection of intervals, merge any overlapping intervals.**

### Example 1:

Input: intervals =  $[[1,3],[2,6],[8,10],[15,18]]$

Output:  $[[1,6],[8,10],[15,18]]$

Explanation: Since intervals  $[1,3]$  and  $[2,6]$  overlap, we merge them into  $[1,6]$ .

### Solution:

```
var merge = function (intervals) {
    if (intervals.length <= 1) return intervals;

    // Step 1: Sort intervals by their start time
    intervals.sort((a, b) => a[0] - b[0]);

    let result = [];
    result.push(intervals[0]);

    // Step 2: Iterate through the intervals and merge them if necessary
    for (let i = 1; i < intervals.length; i++) {
        let last = result[result.length - 1];
        let current = intervals[i];

        // If the current interval overlaps with the last interval in result
        if (last[1] >= current[0]) {
            // Merge the intervals by updating the end time of the last interval
            last[1] = Math.max(last[1], current[1]);
        } else {
            // If no overlap, add the current interval to result
            result.push(current);
        }
    }
}
```

```
    }

    return result;
};
```

### Explanation:

1. **Sort the intervals:**
  - o Sorting the intervals by their start time helps us to easily detect overlapping intervals by just looking at adjacent intervals.
2. **Initialize an empty result array:**
  - o We begin by adding the first interval from the sorted list to the result array, as this interval will serve as the base for merging.
3. **Iterate through the intervals:**
  - o For each interval, we compare it with the last interval added to the result:
    - If the current interval starts after the last interval ends, we simply add the current interval to the result array.
    - If the current interval overlaps with the last interval (i.e., its start time is less than or equal to the end time of the last interval), we merge them by updating the end time of the last interval in the result array.
4. **Return the merged intervals:**
  - o Once all the intervals have been processed, the result array will contain the merged intervals.

### Example Walkthrough:

**Input:** [[1,3],[2,6],[8,10],[15,18]]

1. **Sort the intervals:** [[1,3],[2,6],[8,10],[15,18]]
2. **Start with an empty result array:** []
3. **Process the first interval** [1,3], add it to the result: [[1,3]]
4. **Process the second interval** [2,6], it overlaps with [1,3], so merge them into [1,6]: [[1,6]]
5. **Process the third interval** [8,10], no overlap with [1,6], add [8,10]: [[1,6],[8,10]]
6. **Process the fourth interval** [15,18], no overlap with [8,10], add [15,18]: [[1,6],[8,10],[15,18]]

**Output:** [[1,6],[8,10],[15,18]]

### Merge Intervals

#### 532. Problem:

**Given a collection of intervals, merge all overlapping intervals.**

**You need to merge all the intervals that have any overlap and return the resulting list.**

#### Example 1:

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlap, we merge them into [1,6].

### Solution:

```
var merge = function (intervals) {
    if (intervals.length <= 1) return intervals;

    // Step 1: Sort the intervals by their start time
    intervals.sort((a, b) => a[0] - b[0]);

    let result = [];
    result.push(intervals[0]);

    // Step 2: Iterate through the intervals and merge them if necessary
    for (let i = 1; i < intervals.length; i++) {
        let last = result[result.length - 1];
        let current = intervals[i];

        // If the current interval overlaps with the last interval in result
        if (last[1] >= current[0]) {
            // Merge the intervals by updating the end time of the last interval
            last[1] = Math.max(last[1], current[1]);
        } else {
            // If no overlap, add the current interval to result
            result.push(current);
        }
    }

    return result;
};
```

### Explanation:

1. **Sort the intervals:** We sort the intervals by their start time to make sure we can compare intervals in order. Sorting simplifies the merging process because overlapping intervals will always be adjacent.
2. **Initialize the result array:** Start with the first interval in the sorted list. This will act as the base interval for merging.
3. **Iterate through the sorted intervals:**
  - o For each new interval, check if it overlaps with the last interval in the result array:
    - If they overlap, merge them by updating the end time of the last interval in the result array.
    - If they don't overlap, add the current interval to the result array.
4. **Return the merged intervals:** Once all intervals are processed, the result array will contain the merged intervals.

## Non-overlapping Intervals

**533. Problem:** Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

**Example 1:**

Input: intervals = [[1,2],[2,3],[3,4],[1,3]]

Output: 1

Explanation: The intervals [1,3] and [2,3] overlap, so we remove [2,3].

**Solution:**

```
var eraseOverlapIntervals = function (intervals) {  
    if (intervals.length === 0) return 0;  
  
    // Step 1: Sort the intervals by their end times  
    intervals.sort((a, b) => a[1] - b[1]);  
  
    let count = 0;  
    let lastEnd = intervals[0][1];  
  
    // Step 2: Iterate through the intervals and count the overlaps  
    for (let i = 1; i < intervals.length; i++) {  
        let currentInterval = intervals[i];  
  
        // If the current interval overlaps with the last included one  
        if (currentInterval[0] < lastEnd) {  
            count++; // Remove this interval  
        } else {  
            lastEnd = currentInterval[1]; // Update the end to the current interval's end  
        }  
    }  
  
    return count;  
};
```

**Explanation:**

1. **Sort the intervals:** We start by sorting the intervals based on their end times. This is important because we want to prioritize intervals that finish earlier, which leaves more space for subsequent intervals.
2. **Track the last interval's end time:** We initialize a variable lastEnd to keep track of the end time of the last non-overlapping interval that we've included in the result. Initially, this will be the end time of the first interval (after sorting).
3. **Iterate through the intervals:**

- For each interval, check if it overlaps with the last included interval (i.e., if its start time is less than lastEnd).
  - If there's an overlap, increment the count of removed intervals.
  - If there's no overlap, update the lastEnd to the end time of the current interval.
4. **Return the count of removed intervals:** The counter count will store the number of intervals that need to be removed to make the rest of the intervals non-overlapping.

### Example Walkthrough:

**Input:** [[1,2], [2,3], [3,4], [1,3]]

1. **Sort the intervals:** [[1,2], [1,3], [2,3], [3,4]]
2. **Start with the first interval** [1,2] and set lastEnd = 2.
3. **Process the second interval** [1,3]:
  - Since it starts at 1, which is less than lastEnd = 2, it overlaps with the previous interval. Increment count to 1.
4. **Process the third interval** [2,3]:
  - It does not overlap with the last included interval [1,2] because it starts at 2. Update lastEnd to 3.
5. **Process the fourth interval** [3,4]:
  - It does not overlap with the last included interval [2,3] because it starts at 3. Update lastEnd to 4.
6. **Final result:** The count of removed intervals is 1.

**Output:** 1

### Non-overlapping Intervals

**534. Problem: Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.**

#### Example 1:

Input: intervals = [[1,2],[2,3],[3,4],[1,3]]

Output: 1

Explanation: The intervals [1,3] and [2,3] overlap, so we remove [2,3].

#### Solution:

```
var eraseOverlapIntervals = function (intervals) {
  if (intervals.length === 0) return 0;

  // Sort the intervals by their end times
  intervals.sort((a, b) => a[1] - b[1]);

  let count = 0;
  let lastEnd = intervals[0][1];

  // Iterate through the intervals and count the overlaps
  for (let i = 1; i < intervals.length; i++) {
```

```

let currentInterval = intervals[i];

// If the current interval overlaps with the last included one
if (currentInterval[0] < lastEnd) {
    count++; // Remove this interval
} else {
    lastEnd = currentInterval[1]; // Update the end to the current interval's end
}
}

return count;
};

```

### **Explanation:**

1. **Sort the intervals:** Sorting by the end times helps prioritize intervals that take the least space. This is key to minimizing the number of intervals we need to remove.
2. **Track the end time of the last included interval:** Start with the first interval and set its end time as lastEnd. This will serve as a reference to check if future intervals overlap.
3. **Iterate through the sorted intervals:**
  - o For each interval, check if its start time is less than lastEnd. If it overlaps, we remove it (increment count).
  - o If it doesn't overlap, we update lastEnd to the end time of the current interval.
4. **Return the count:** The result is the number of intervals we had to remove to make the remaining intervals non-overlapping.

### **Example Walkthrough:**

**Input:** [[1,2], [2,3], [3,4], [1,3]]

1. **Sort the intervals:** [[1,2], [1,3], [2,3], [3,4]]
2. **Initialize:** count = 0, lastEnd = 2 (end time of the first interval [1,2])
3. **Iterate through the intervals:**
  - o Interval [1,3]: overlaps with [1,2] (since 1 < 2), so remove it (count = 1).
  - o Interval [2,3]: does not overlap with [1,2] (update lastEnd = 3).
  - o Interval [3,4]: does not overlap with [2,3] (update lastEnd = 4).
4. **Return the count:** count = 1.

Thus, the minimum number of intervals to remove is 1.

### **Non-overlapping Intervals**

**535. Problem:** Given a collection of intervals, return the maximum number of intervals you can remove to make the rest of the intervals non-overlapping.

**Example 1:**

**Input:** intervals = [[1,2],[2,3]]

Output: 0

Explanation: No need to remove any intervals as they do not overlap.

### Solution:

```
var eraseOverlapIntervals = function (intervals) {
    if (intervals.length === 0) return 0;

    // Sort intervals by their end times
    intervals.sort((a, b) => a[1] - b[1]);

    let count = 0;
    let lastEnd = intervals[0][1];

    // Iterate through intervals and check overlaps
    for (let i = 1; i < intervals.length; i++) {
        let currentInterval = intervals[i];

        // If the current interval overlaps, remove it
        if (currentInterval[0] < lastEnd) {
            count++;
        } else {
            lastEnd = currentInterval[1]; // Update end time if no overlap
        }
    }

    return count;
};
```

### Explanation:

1. **Sorting the intervals:** The greedy approach is based on sorting the intervals by their end time. The rationale is that intervals that finish earlier leave more room for future intervals. Therefore, by sorting by end time, we can maximize the number of intervals we keep.
2. **Tracking the last end time:** After sorting, we initialize lastEnd with the end time of the first interval. As we move through the intervals, we compare the start time of the current interval with lastEnd.
3. **Removing intervals that overlap:** If the start time of the current interval is less than lastEnd, we increment the counter (count) to signify that we are removing the current interval because it overlaps with the last non-overlapping interval.
4. **Updating the last end time:** If the current interval does not overlap with the last chosen interval, we update lastEnd to the end time of the current interval.
5. **Returning the result:** After iterating through all intervals, the count variable will contain the number of intervals that need to be removed to make the rest non-overlapping.

## Example Walkthrough:

**Input:** [[1,2], [2,3], [3,4], [1,3]]

1. **Sort the intervals:** [[1, 2], [1, 3], [2, 3], [3, 4]]
2. **Initialize:** count = 0, lastEnd = 2 (end time of the first interval [1,2]).
3. **Iterate through the intervals:**
  - o Interval [1, 3]: overlaps with [1, 2] (since 1 < 2), so remove it (count = 1).
  - o Interval [2, 3]: does not overlap with [1, 2], so update lastEnd = 3.
  - o Interval [3, 4]: does not overlap with [2, 3], so update lastEnd = 4.
4. **Final result:** The number of intervals removed is 1.

## Final Answer:

- **Output:** 1

## Non-overlapping Intervals

**536. Problem:** Given a collection of intervals, return the maximum number of intervals you can remove to make the rest of the intervals non-overlapping.

### Example 1:

Input: intervals = [[1,2],[1,2],[1,2]]

Output: 2

Explanation: You need to remove two [1,2] intervals to make the remaining interval non-overlapping.

### Solution:

```
var eraseOverlapIntervals = function (intervals) {  
    if (intervals.length === 0) return 0;  
  
    // Sort intervals by their end times  
    intervals.sort((a, b) => a[1] - b[1]);  
  
    let count = 0;  
    let lastEnd = intervals[0][1];  
  
    // Iterate through intervals and check overlaps  
    for (let i = 1; i < intervals.length; i++) {  
        let currentInterval = intervals[i];  
  
        // If the current interval overlaps, remove it  
        if (currentInterval[0] < lastEnd) {  
            count++;  
        } else {  
            lastEnd = currentInterval[1]; // Update end time if no overlap  
        }  
    }  
}
```

```
    return count;
};
```

### Explanation:

1. **Sorting the intervals:** The greedy approach is based on sorting the intervals by their end time. The rationale is that intervals that finish earlier leave more room for future intervals. Therefore, by sorting by end time, we can maximize the number of intervals we keep.
2. **Tracking the last end time:** After sorting, we initialize lastEnd with the end time of the first interval. As we move through the intervals, we compare the start time of the current interval with lastEnd.
3. **Removing intervals that overlap:** If the start time of the current interval is less than lastEnd, we increment the counter (count) to signify that we are removing the current interval because it overlaps with the last non-overlapping interval.
4. **Updating the last end time:** If the current interval does not overlap with the last chosen interval, we update lastEnd to the end time of the current interval.
5. **Returning the result:** After iterating through all intervals, the count variable will contain the number of intervals that need to be removed to make the rest non-overlapping.

### Example Walkthrough:

**Input:** [[1,2], [2,3], [3,4], [1,3]]

1. **Sort the intervals:** [[1, 2], [1, 3], [2, 3], [3, 4]]
2. **Initialize:** count = 0, lastEnd = 2 (end time of the first interval [1,2]).
3. **Iterate through the intervals:**
  - o Interval [1, 3]: overlaps with [1, 2] (since 1 < 2), so remove it (count = 1).
  - o Interval [2, 3]: does not overlap, so update lastEnd = 3.
  - o Interval [3, 4]: does not overlap, so update lastEnd = 4.
4. **Return result:** The result is 1 because we removed one overlapping interval.

### Time Complexity:

- Sorting takes **O(n log n)**.
- Iterating through the intervals takes **O(n)**.
- **Overall time complexity:** **O(n log n)**.

### Space Complexity:

- Sorting requires **O(n)** space.
- **Space complexity:** **O(n)**.

## Non-overlapping Intervals

**537. Problem: Given a collection of intervals, find the maximum number of non-overlapping intervals you can select.**

**Example 1:**

Input: intervals = [[1,2],[2,3]]

Output: 2

Explanation: Both intervals do not overlap.

**Solution:**

```
var eraseOverlapIntervals = function (intervals) {  
    if (intervals.length === 0) return 0;  
  
    // Sort intervals by their end times  
    intervals.sort((a, b) => a[1] - b[1]);  
  
    let count = 0;  
    let lastEnd = intervals[0][1];  
  
    // Iterate through intervals and check overlaps  
    for (let i = 1; i < intervals.length; i++) {  
        let currentInterval = intervals[i];  
  
        // If the current interval overlaps, remove it  
        if (currentInterval[0] < lastEnd) {  
            count++;  
        } else {  
            lastEnd = currentInterval[1]; // Update end time if no overlap  
        }  
    }  
  
    return count;  
};
```

**Explanation:**

1. **Sorting the intervals:** The idea behind sorting is to always select intervals that end earlier to maximize the number of intervals we can keep. Once sorted, the intervals are processed in order of their end times.
2. **Tracking the last end time:** As we iterate through the intervals, we maintain the lastEnd variable to track the end time of the last selected non-overlapping interval. This ensures that the current interval does not overlap with the previous interval.
3. **Selecting non-overlapping intervals:** If the start time of the current interval is greater than or equal to lastEnd, then the current interval doesn't overlap with the previous one, and we can include it in our solution. If there's an overlap, we increment the count of intervals to be removed.
4. **Returning the result:** The total number of intervals that need to be removed is stored in the count variable, and it is returned as the result.

### Example Walkthrough:

**Input:** [[1,2], [2,3], [3,4], [1,3]]

1. **Sort the intervals:** [[1, 2], [1, 3], [2, 3], [3, 4]]
2. **Initialize:** count = 0, lastEnd = 2 (end time of the first interval [1, 2]).
3. **Iterate through the intervals:**
  - o Interval [1, 3]: overlaps with [1, 2] (since 1 < 2), so remove it (count = 1).
  - o Interval [2, 3]: does not overlap with [1, 2], update lastEnd = 3 and move to the next interval.
  - o Interval [3, 4]: does not overlap with [2, 3], update lastEnd = 4 and move to the next interval.
4. **Result:** We have selected 3 intervals: [1, 2], [2, 3], and [3, 4].

Thus, the output will be **3** as we can select 3 non-overlapping intervals.

## 8. Matrix & Grid-Based Challenges

### Setting Zero Values in Matrices Efficiently

**538. Problem:** Given an m x n matrix, if an element in the matrix is 0, set its entire row and column to 0. Do it in place.

**Example 1:**

Input: matrix = [  
[1, 1, 1],  
[1, 0, 1],  
[1, 1, 1],  
];

Output: [  
[1, 0, 1],  
[0, 0, 0],  
[1, 0, 1],  
];

**Solution:**

```
var setZeroes = function (matrix) {  
    let m = matrix.length;  
    let n = matrix[0].length;  
    let isFirstRowZero = false;  
    let isFirstColumnZero = false;  
  
    // Check if the first row contains a zero  
    for (let j = 0; j < n; j++) {  
        if (matrix[0][j] === 0) {  
            isFirstRowZero = true;  
            break;  
        }  
    }  
  
    // Check if the first column contains a zero  
    for (let i = 1; i < m; i++) {  
        if (matrix[i][0] === 0) {  
            isFirstColumnZero = true;  
            break;  
        }  
    }  
  
    // Set the rest of the matrix to zero  
    for (let i = 1; i < m; i++) {  
        for (let j = 1; j < n; j++) {  
            if (isFirstRowZero && matrix[i][j] === 1) {  
                matrix[i][j] = 0;  
            } else if (isFirstColumnZero && matrix[i][j] === 1) {  
                matrix[i][j] = 0;  
            } else if (matrix[i][j] === 0) {  
                matrix[i][j] = 0;  
            }  
        }  
    }  
}
```

```

        }

    // Check if the first column contains a zero
    for (let i = 0; i < m; i++) {
        if (matrix[i][0] === 0) {
            isFirstColumnZero = true;
            break;
        }
    }

    // Mark the first row and column for setting zeros
    for (let i = 1; i < m; i++) {
        for (let j = 1; j < n; j++) {
            if (matrix[i][j] === 0) {
                matrix[i][0] = 0;
                matrix[0][j] = 0;
            }
        }
    }

    // Set zeros based on markers in the first row and column
    for (let i = 1; i < m; i++) {
        for (let j = 1; j < n; j++) {
            if (matrix[i][0] === 0 || matrix[0][j] === 0) {
                matrix[i][j] = 0;
            }
        }
    }

    // Set the first row to zero if needed
    if (isFirstRowZero) {
        for (let j = 0; j < n; j++) {
            matrix[0][j] = 0;
        }
    }

    // Set the first column to zero if needed
    if (isFirstColumnZero) {
        for (let i = 0; i < m; i++) {
            matrix[i][0] = 0;
        }
    }
};

```

### Explanation:

1. Identify if first row and first column should be zeroed:

- We first check if there's any zero in the first row or column. This is necessary because we will use the first row and column to mark other rows and columns that need to be zeroed.
- 2. Mark rows and columns:**
    - Starting from the second row and second column, we check each element. If we find a 0, we mark the first row and first column of that element's row and column by setting them to 0.
  - 3. Set the elements to zero:**
    - For each element starting from the second row and column, we check if its row or column is marked as 0. If it is, we set that element to 0.
  - 4. Handle the first row and column separately:**
    - After the second pass, we handle the first row and column. If they were marked as needing to be zeroed (from our initial check), we set the entire first row or column to 0.
  - 5. Final result:**
    - The matrix is now updated in place, with all necessary rows and columns set to zero.

## **01 Matrix**

### **539. Problem:**

**Given an  $m \times n$  binary matrix filled with 0's and 1's, find the distance of the nearest 0 for each cell.**

**The distance between two adjacent cells is 1.**

#### **Example 1:**

```
Input: matrix = [
  [0, 0, 0],
  [0, 1, 0],
  [0, 0, 0],
];
```

```
Output: [
  [0, 0, 0],
  [0, 1, 0],
  [0, 0, 0],
];
```

#### **Solution:**

```
var updateMatrix = function (matrix) {
  let m = matrix.length;
  let n = matrix[0].length;
  let queue = [];
  let directions = [
```

```

[0, 1],
[1, 0],
[0, -1],
[-1, 0],
];

// Initialize the result matrix and queue
for (let i = 0; i < m; i++) {
  for (let j = 0; j < n; j++) {
    if (matrix[i][j] === 0) {
      queue.push([i, j]);
    } else {
      matrix[i][j] = Infinity;
    }
  }
}

// Perform BFS
while (queue.length > 0) {
  let [x, y] = queue.shift();

  // Check all four directions
  for (let [dx, dy] of directions) {
    let nx = x + dx,
        ny = y + dy;

    // Check if the new coordinates are within bounds and can be updated
    if (
      nx >= 0 &&
      nx < m &&
      ny >= 0 &&
      ny < n &&
      matrix[nx][ny] > matrix[x][y] + 1
    ) {
      matrix[nx][ny] = matrix[x][y] + 1;
      queue.push([nx, ny]);
    }
  }
}

return matrix;
};

```

### Explanation:

#### 1. Initial Setup:

- We iterate through the matrix. All cells containing 0 are added to the queue because they are the starting points for BFS.
  - For cells with 1, we initialize their distance to infinity since we haven't calculated it yet.
2. **Breadth-First Search (BFS):**
    - We dequeue a cell and check its four neighbors (up, down, left, right). If a neighbor has a greater distance value, we update it to be the current cell's distance + 1.
    - After updating the neighbor, we enqueue it for future exploration.
  3. **Final Matrix:**
    - After BFS completes, all cells will have their correct distance to the nearest 0.

## Walls and Gates

### 540. Problem:

You are given an  $m \times n$  2D grid, and each cell can have one of the following values:

- -1 representing a wall,
- 0 representing a gate,
- INF representing an empty room.

You need to fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, the room should remain filled with INF.

Write a solution that updates the grid in-place.

### Example 1:

```
Input: grid = [
    [INF, -1, 0, INF],
    [INF, INF, INF, -1],
    [INF, -1, INF, -1],
    [0, -1, INF, INF],
];
```

```
Output: grid = [
    [3, -1, 0, 1],
    [2, 2, 1, -1],
    [1, -1, 2, -1],
    [0, -1, 3, 4],
];
```

### Solution:

```
var wallsAndGates = function (grid) {
    let m = grid.length;
    let n = grid[0].length;
```

```

let queue = [];
let directions = [
  [0, 1],
  [1, 0],
  [0, -1],
  [-1, 0],
];

// Initialize the queue with all gates' positions (0's)
for (let i = 0; i < m; i++) {
  for (let j = 0; j < n; j++) {
    if (grid[i][j] === 0) {
      queue.push([i, j]);
    }
  }
}

// Perform BFS
while (queue.length > 0) {
  let [x, y] = queue.shift();

  // Check all four directions
  for (let [dx, dy] of directions) {
    let nx = x + dx,
        ny = y + dy;

    // Check if the new coordinates are within bounds and can be updated
    if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] === Infinity) {
      grid[nx][ny] = grid[x][y] + 1;
      queue.push([nx, ny]);
    }
  }
}
};

```

## Explanation:

### 1. Initial Setup:

- o We first iterate through the grid to find all gates (0s). Each gate is enqueued into the BFS queue.
- o All INF cells are left as they are, indicating that they haven't been visited yet.

### 2. Breadth-First Search (BFS):

- o Starting from all the gates, we perform BFS to propagate the distances to all reachable rooms.
- o The first time we visit an empty room (INF), we update its distance to be the current cell's distance + 1 (since each move represents a distance of 1).

- We continue this process for all cells until all rooms that can be reached from a gate have been updated with their minimum distance.

### 3. Final Grid:

- After BFS completes, the grid will be updated with the shortest distance to a gate for every empty room, and walls will remain unchanged.

#### Time Complexity:

- $O(m * n)$  — We visit each cell at most once in BFS.

#### Space Complexity:

- $O(m * n)$  — The queue can store all cells in the worst case.

### Traversing a Spiral Matrix II

#### 541. Problem:

**Given a positive integer n, generate an n x n matrix filled with elements from 1 to  $n^2$  in spiral order.**

#### Example 1:

Input:  $n = 3$ ;

Output: [  
 [1, 2, 3],  
 [8, 9, 4],  
 [7, 6, 5],  
 ];

#### Solution:

```
var generateMatrix = function (n) {
  let result = new Array(n).fill().map(() => new Array(n).fill(0));
  let top = 0,
    bottom = n - 1,
    left = 0,
    right = n - 1;
  let num = 1;

  while (top <= bottom && left <= right) {
    // Traverse from left to right on the top row
    for (let i = left; i <= right; i++) {
      result[top][i] = num++;
    }
    top++;
    // Traverse downwards on the right column
    for (let i = top; i <= bottom; i++) {
      result[i][right] = num++;
    }
    right--;
    // Traverse from right to left on the bottom row
    for (let i = right; i >= left; i--) {
      result[bottom][i] = num++;
    }
    bottom--;
    // Traverse upwards on the left column
    for (let i = bottom; i >= top; i--) {
      result[i][left] = num++;
    }
    left++;
  }
  return result;
}
```

```

    }

    right--;
}

// Traverse from right to left on the bottom row
if (top <= bottom) {
    for (let i = right; i >= left; i--) {
        result[bottom][i] = num++;
    }
    bottom--;
}

// Traverse upwards on the left column
if (left <= right) {
    for (let i = bottom; i >= top; i--) {
        result[i][left] = num++;
    }
    left++;
}
}

return result;
};

```

### Explanation:

1. **Initialization:**
  - o We start by initializing the result matrix, `result`, which is a 2D array of size  $n \times n$ , filled with zeros.
  - o We also define the boundaries for the spiral traversal: `top`, `bottom`, `left`, and `right`. Initially, `top` is 0, `bottom` is  $n - 1$ , `left` is 0, and `right` is  $n - 1$ .
2. **Filling the Matrix:**
  - o We start filling the matrix from 1 up to  $n^2$ .
  - o We move in a clockwise spiral direction:
    - **Right across the top row:** Start from the left boundary and move towards the right boundary.
    - **Down along the right column:** After completing the top row, move down the right column from top to bottom.
    - **Left across the bottom row:** Once reaching the bottom row, move left from right to left.
    - **Up along the left column:** After completing the bottom row, move up the left column from bottom to top.
  - o After each move, we adjust the boundaries inward to fill the next set of cells in a spiral.
3. **Termination Condition:**
  - o The loop terminates when the `top` boundary exceeds `bottom`, or the `left` boundary exceeds `right`. At that point, the entire matrix has been filled.
4. **Return the Result:**
  - o After completing the spiral traversal, the filled matrix is returned as the output.

## 9. Heaps & Searching, Sorting

### Merging Multiple Sorted Data Sets (Using a Min-Heap)

#### 542. Problem:

Given k sorted linked lists, merge them into one sorted linked list and return it.

#### Example 1:

Input:

[1->4->5, 1->3->4, 2->6]

Output:

1->1->2->3->4->4->5->6

#### Solution:

```
function mergeKLists(lists) {
    const heap = new MinHeap();
    const dummyHead = new ListNode(0);
    let current = dummyHead;

    // Initialize the heap with the head of each list
    for (let list of lists) {
        if (list) {
            heap.push(list);
        }
    }

    // Extract the minimum element and push the next element of the same list into the heap
    while (heap.size() > 0) {
        let node = heap.pop();
        current.next = node;
        current = current.next;

        // If there is a next node in the same list, push it into the heap
        if (node.next) {
            heap.push(node.next);
        }
    }

    return dummyHead.next;
}

// MinHeap class to manage heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }

    push(item) {
        this.heap.push(item);
        this.bubbleUp(this.heap.length - 1);
    }

    pop() {
        if (this.heap.length === 0) return null;
        const item = this.heap[0];
        const lastItem = this.heap.pop();
        if (lastItem) {
            this.heap[0] = lastItem;
            this.bubbleDown(0);
        }
        return item;
    }

    bubbleUp(index) {
        let parentIndex = Math.floor((index - 1) / 2);
        while (parentIndex >= 0 && this.heap[parentIndex].value > this.heap[index].value) {
            [this.heap[parentIndex], this.heap[index]] = [this.heap[index], this.heap[parentIndex]];
            index = parentIndex;
            parentIndex = Math.floor((index - 1) / 2);
        }
    }

    bubbleDown(index) {
        let leftChildIndex = index * 2 + 1;
        let rightChildIndex = index * 2 + 2;
        let smallestIndex = index;

        if (leftChildIndex < this.heap.length && this.heap[leftChildIndex].value < this.heap[smallestIndex].value) {
            smallestIndex = leftChildIndex;
        }

        if (rightChildIndex < this.heap.length && this.heap[rightChildIndex].value < this.heap[smallestIndex].value) {
            smallestIndex = rightChildIndex;
        }

        if (smallestIndex !== index) {
            [this.heap[index], this.heap[smallestIndex]] = [this.heap[smallestIndex], this.heap[index]];
            this.bubbleDown(smallestIndex);
        }
    }

    size() {
        return this.heap.length;
    }
}
```

```

}

// Insert node into heap
push(node) {
  this.heap.push(node);
  this._heapifyUp();
}

// Pop the minimum node
pop() {
  if (this.heap.length === 0) return null;

  if (this.heap.length === 1) return this.heap.pop();

  let root = this.heap[0];
  this.heap[0] = this.heap.pop();
  this._heapifyDown();
  return root;
}

// Helper functions for heap operations
_heapifyUp() {
  let index = this.heap.length - 1;
  while (index > 0) {
    let parentIndex = Math.floor((index - 1) / 2);
    if (this.heap[index].val >= this.heap[parentIndex].val) break;
    [this.heap[index], this.heap[parentIndex]] = [
      this.heap[parentIndex],
      this.heap[index],
    ];
    index = parentIndex;
  }
}

_heapifyDown() {
  let index = 0;
  const length = this.heap.length;
  while (index < length) {
    let leftChildIndex = 2 * index + 1;
    let rightChildIndex = 2 * index + 2;
    let smallest = index;

    if (
      leftChildIndex < length &&
      this.heap[leftChildIndex].val < this.heap[smallest].val
    ) {
      smallest = leftChildIndex;
    }
  }
}

```

```

if (
    rightChildIndex < length &&
    this.heap[rightChildIndex].val < this.heap[smallest].val
) {
    smallest = rightChildIndex;
}

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [
    this.heap[smallest],
    this.heap[index],
];
index = smallest;
}
}

size() {
    return this.heap.length;
}
}

```

### **Explanation:**

#### **1. Min-Heap Initialization:**

- A MinHeap class is used to handle the insertion and extraction of nodes. The heap is structured so that the smallest element can be accessed in constant time.
- We initialize the heap by pushing the head node from each linked list into the heap. This gives us access to the smallest element from all the lists.

#### **2. Heap Operations:**

- The heap ensures that each time we extract a node, it is the smallest node available across all lists.
- After extracting a node from the heap, we add its next node (if it exists) into the heap, ensuring that we maintain the smallest element at the top.

#### **3. Merging Lists:**

- We start with a dummy node to simplify the logic of merging the nodes. As we pop nodes from the heap, we append them to the result list.
- The result list is constructed in a sorted order since we always pick the smallest node.

#### **4. Efficiency:**

- The Min-Heap ensures that at each step, we are efficiently picking the smallest node from the k lists, which allows us to merge the lists in optimal time.

### **Merging Multiple Sorted Data Sets (Using Min-Heap)**

### 543. Problem:

Given k sorted linked lists, merge them into one sorted linked list and return it.

#### Example 1:

Input:

[1->4->5, 1->3->4, 2->6]

Output:

1->1->2->3->4->4->5->6

#### Solution:

```
function mergeKLists(lists) {
    const heap = new MinHeap();
    const dummyHead = new ListNode(0);
    let current = dummyHead;

    // Initialize the heap with the head of each list
    for (let list of lists) {
        if (list) {
            heap.push(list);
        }
    }

    // Extract the minimum element and push the next element of the same list into the heap
    while (heap.size() > 0) {
        let node = heap.pop();
        current.next = node;
        current = current.next;

        // If there is a next node in the same list, push it into the heap
        if (node.next) {
            heap.push(node.next);
        }
    }

    return dummyHead.next;
}

// MinHeap class to manage heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }

    // Insert node into heap
    push(node) {
```

```
this.heap.push(node);
this._heapifyUp();
}

// Pop the minimum node
pop() {
if (this.heap.length === 0) return null;

if (this.heap.length === 1) return this.heap.pop();

let root = this.heap[0];
this.heap[0] = this.heap.pop();
this._heapifyDown();
return root;
}

// Helper functions for heap operations
_heapifyUp() {
let index = this.heap.length - 1;
while (index > 0) {
let parentIndex = Math.floor((index - 1) / 2);
if (this.heap[index].val >= this.heap[parentIndex].val) break;
[this.heap[index], this.heap[parentIndex]] = [
this.heap[parentIndex],
this.heap[index],
];
index = parentIndex;
}
}

_heapifyDown() {
let index = 0;
const length = this.heap.length;
while (index < length) {
let leftChildIndex = 2 * index + 1;
let rightChildIndex = 2 * index + 2;
let smallest = index;

if (
leftChildIndex < length &&
this.heap[leftChildIndex].val < this.heap[smallest].val
) {
smallest = leftChildIndex;
}

if (
rightChildIndex < length &&
this.heap[rightChildIndex].val < this.heap[smallest].val
)
```

```

) {
    smallest = rightChildIndex;
}

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [
    this.heap[smallest],
    this.heap[index],
];
index = smallest;
}

size() {
    return this.heap.length;
}
}

```

### Explanation:

#### 1. Min-Heap Initialization:

- We initialize the heap with the first node from each linked list. The heap will automatically keep track of the smallest value among the first nodes.

#### 2. Merging Process:

- At each step, we extract the smallest node from the heap, which is guaranteed to be the next smallest element in the merged list.
- If the extracted node has a next node in its list, that next node is inserted into the heap, ensuring that the heap always contains the next smallest node from each list.

#### 3. Final Merged List:

- The result is built by appending nodes to a dummy head node, and at the end, we return the merged list starting from the next of the dummy head.

#### 4. Min-Heap Operations:

- The heap ensures that we always get the smallest node efficiently in **O(log k)** time. The insertion and extraction of nodes from the heap are the key operations that drive the merging process.

### Time Complexity:

- Inserting a node into the heap takes **O(log k)** time, and we perform this operation for each of the n nodes.
- Thus, the time complexity for this algorithm is **O(n log k)**, where n is the total number of nodes in all lists, and k is the number of lists.

## Space Complexity:

- The heap stores at most k elements at any point, which results in a space complexity of **O(k)**. This is because we are storing the current node of each linked list in the heap at any given time.

## Merging Multiple Sorted Data Sets (Using Heap)

**544. Problem: Given k sorted linked lists, merge them into one sorted linked list.**

**Example 1:**

Input:

[1->4->5, 1->3->4, 2->6]

Output:

1->1->2->3->4->4->5->6

**Solution:**

```
function mergeKLists(lists) {
    const heap = new MinHeap();
    const dummyHead = new ListNode(0);
    let current = dummyHead;

    // Insert the first node of each list into the heap
    for (let list of lists) {
        if (list) {
            heap.push(list);
        }
    }

    // While the heap is not empty, extract the minimum node and push the next node if it exists
    while (heap.size() > 0) {
        let node = heap.pop();
        current.next = node;
        current = current.next;

        // If there is a next node in the same list, push it into the heap
        if (node.next) {
            heap.push(node.next);
        }
    }

    return dummyHead.next;
}

// MinHeap class to manage heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }
}
```

```

// Insert node into heap
push(node) {
    this.heap.push(node);
    this._heapifyUp();
}

// Pop the minimum node
pop() {
    if (this.heap.length === 0) return null;

    if (this.heap.length === 1) return this.heap.pop();

    let root = this.heap[0];
    this.heap[0] = this.heap.pop();
    this._heapifyDown();
    return root;
}

// Helper functions for heap operations
_heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
        let parentIndex = Math.floor((index - 1) / 2);
        if (this.heap[index].val >= this.heap[parentIndex].val) break;
        [this.heap[index], this.heap[parentIndex]] = [
            this.heap[parentIndex],
            this.heap[index],
        ];
        index = parentIndex;
    }
}

_heapifyDown() {
    let index = 0;
    const length = this.heap.length;
    while (index < length) {
        let leftChildIndex = 2 * index + 1;
        let rightChildIndex = 2 * index + 2;
        let smallest = index;

        if (
            leftChildIndex < length &&
            this.heap[leftChildIndex].val < this.heap[smallest].val
        ) {
            smallest = leftChildIndex;
        }
    }
}

```

```

if (
    rightChildIndex < length &&
    this.heap[rightChildIndex].val < this.heap[smallest].val
) {
    smallest = rightChildIndex;
}

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [
    this.heap[smallest],
    this.heap[index],
];
index = smallest;
}
}

size() {
    return this.heap.length;
}
}

```

### **Explanation of Code:**

#### **1. Heap Initialization:**

- We use a custom MinHeap class to handle the heap operations. The heap stores nodes from the linked lists.
- The push and pop methods ensure that the heap maintains the min-heap property, i.e., the smallest value is always at the root.

#### **2. Inserting and Removing Nodes:**

- For each node, we first insert the initial nodes of all the lists into the heap. Then, we repeatedly extract the smallest node from the heap and add it to the result list.
- If the extracted node has a next node, it is inserted back into the heap, ensuring that the heap always contains the smallest nodes at the top.

#### **3. Result Construction:**

- We use a dummy head node to simplify the process of building the resulting linked list.

### **Time and Space Complexity:**

- **Time Complexity:**  $O(n \log k)$ , where  $n$  is the total number of nodes across all lists, and  $k$  is the number of lists.
- **Space Complexity:**  $O(k)$ , due to the heap storing at most  $k$  nodes at any time.

### **Merging Multiple Sorted Data Sets (Using Min-Heap)**

**545. Problem: You are given k sorted linked lists, and your task is to merge them into one sorted linked list. Use a Min-Heap to solve the problem efficiently.**

**Example:**

Input:  
[1->4->5, 1->3->4, 2->6]

Output:  
1->1->2->3->4->4->5->6

**Solution:**

```
function mergeKLists(lists) {
    const heap = new MinHeap();
    const dummyHead = new ListNode(0);
    let current = dummyHead;

    // Insert the first node of each list into the heap
    for (let list of lists) {
        if (list) {
            heap.push(list);
        }
    }

    // While the heap is not empty, extract the minimum node and push the next node if it exists
    while (heap.size() > 0) {
        let node = heap.pop();
        current.next = node;
        current = current.next;

        // If there is a next node in the same list, push it into the heap
        if (node.next) {
            heap.push(node.next);
        }
    }

    return dummyHead.next;
}

// MinHeap class to manage heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }

    // Insert node into heap
    push(node) {
        this.heap.push(node);
        this._heapifyUp();
    }

    _heapifyUp() {
        let index = this.heap.length - 1;
        let parentIndex = Math.floor((index - 1) / 2);

        while (parentIndex >= 0 && this.heap[parentIndex].value > this.heap[index].value) {
            [this.heap[parentIndex], this.heap[index]] = [this.heap[index], this.heap[parentIndex]];
            index = parentIndex;
            parentIndex = Math.floor((index - 1) / 2);
        }
    }

    _heapifyDown(index) {
        let leftChildIndex = index * 2 + 1;
        let rightChildIndex = index * 2 + 2;
        let smallestIndex = index;

        if (leftChildIndex < this.heap.length && this.heap[leftChildIndex].value < this.heap[smallestIndex].value) {
            smallestIndex = leftChildIndex;
        }

        if (rightChildIndex < this.heap.length && this.heap[rightChildIndex].value < this.heap[smallestIndex].value) {
            smallestIndex = rightChildIndex;
        }

        if (smallestIndex !== index) {
            [this.heap[index], this.heap[smallestIndex]] = [this.heap[smallestIndex], this.heap[index]];
            this._heapifyDown(smallestIndex);
        }
    }

    pop() {
        if (this.heap.length === 0) {
            return null;
        }

        let result = this.heap[0];
        let lastNode = this.heap.pop();
        if (this.heap.length === 0) {
            return result;
        }

        this.heap[0] = lastNode;
        this._heapifyDown(0);
        return result;
    }

    size() {
        return this.heap.length;
    }
}
```

```

}

// Pop the minimum node
pop() {
  if (this.heap.length === 0) return null;

  if (this.heap.length === 1) return this.heap.pop();

  let root = this.heap[0];
  this.heap[0] = this.heap.pop();
  this._heapifyDown();
  return root;
}

// Helper functions for heap operations
_heapifyUp() {
  let index = this.heap.length - 1;
  while (index > 0) {
    let parentIndex = Math.floor((index - 1) / 2);
    if (this.heap[index].val >= this.heap[parentIndex].val) break;
    [this.heap[index], this.heap[parentIndex]] = [
      this.heap[parentIndex],
      this.heap[index],
    ];
    index = parentIndex;
  }
}

_heapifyDown() {
  let index = 0;
  const length = this.heap.length;
  while (index < length) {
    let leftChildIndex = 2 * index + 1;
    let rightChildIndex = 2 * index + 2;
    let smallest = index;

    if (
      leftChildIndex < length &&
      this.heap[leftChildIndex].val < this.heap[smallest].val
    ) {
      smallest = leftChildIndex;
    }

    if (
      rightChildIndex < length &&
      this.heap[rightChildIndex].val < this.heap[smallest].val
    ) {
      smallest = rightChildIndex;
    }
  }
}

```

```

    }

    if (smallest === index) break;
    [this.heap[index], this.heap[smallest]] = [
      this.heap[smallest],
      this.heap[index],
    ];
    index = smallest;
  }
}

size() {
  return this.heap.length;
}
}

// ListNode definition
function ListNode(val = 0, next = null) {
  this.val = val;
  this.next = next;
}

```

### **Explanation of the Code:**

#### **1. MergeKLists Function:**

- The function starts by creating a MinHeap and a dummy node (dummyHead) to build the merged list.
- It inserts the head node of each linked list into the heap.
- Then, it processes the heap by repeatedly popping the smallest node, adding it to the merged list, and pushing its next node (if it exists) into the heap.
- Finally, the function returns the merged list, starting from dummyHead.next.

#### **2. MinHeap Class:**

- The MinHeap class implements the heap operations, including insertion (push) and removal (pop).
- The helper functions \_heapifyUp and \_heapifyDown ensure the heap maintains the correct order after each insertion and removal.

#### **3. ListNode:**

- The ListNode class is a simple linked list node class that holds a value and a reference to the next node.

### **Time and Space Complexity:**

- **Time Complexity:**  $O(n \log k)$ , where  $n$  is the total number of nodes across all  $k$  linked lists, and  $k$  is the number of linked lists.
- **Space Complexity:**  $O(k)$ , as the heap stores at most  $k$  nodes at any time.

### **Merging Multiple Sorted Data Sets (Using Heap)**

**546. Problem:** You are given an array of k sorted linked lists. Merge them into a single sorted linked list and return it.

You need to implement an efficient solution that uses a Min-Heap to keep track of the smallest elements.

**Example:**

**Input:**

```
lists = [
  [1, 4, 5],
  [1, 3, 4],
  [2, 6],
];
```

**Output:**

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

**Solution:**

```
class ListNode {
  constructor(val = 0, next = null) {
    this.val = val;
    this.next = next;
  }
}

var mergeKLists = function (lists) {
  const minHeap = new MinHeap();
  const resultHead = new ListNode();
  let current = resultHead;

  // Insert the head of each list into the heap
  for (let i = 0; i < lists.length; i++) {
    if (lists[i]) {
      minHeap.insert(lists[i]);
    }
  }

  // Extract the smallest element and add the next node from the same list
  while (!minHeap.isEmpty()) {
    const node = minHeap.extractMin();
    current.next = node;
    current = current.next;
  }
}
```

```

    if (node.next) {
      minHeap.insert(node.next);
    }
  }

  return resultHead.next;
};

// MinHeap class to support min-heap operations
class MinHeap {
  constructor() {
    this.heap = [];
  }

  insert(node) {
    this.heap.push(node);
    this.heapifyUp();
  }

  extractMin() {
    if (this.isEmpty()) return null;
    const minNode = this.heap[0];
    const lastNode = this.heap.pop();
    if (this.heap.length > 0) {
      this.heap[0] = lastNode;
      this.heapifyDown();
    }
    return minNode;
  }

  heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
      const parentIndex = Math.floor((index - 1) / 2);
      if (this.heap[index].val >= this.heap[parentIndex].val) break;
      [this.heap[index], this.heap[parentIndex]] = [
        this.heap[parentIndex],
        this.heap[index],
      ];
      index = parentIndex;
    }
  }

  heapifyDown() {
    let index = 0;
    const size = this.heap.length;
    while (index < size) {

```

```

const leftChildIndex = 2 * index + 1;
const rightChildIndex = 2 * index + 2;
let smallest = index;

if (
  leftChildIndex < size &&
  this.heap[leftChildIndex].val < this.heap[smallest].val
) {
  smallest = leftChildIndex;
}

if (
  rightChildIndex < size &&
  this.heap[rightChildIndex].val < this.heap[smallest].val
) {
  smallest = rightChildIndex;
}

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [
  this.heap[smallest],
  this.heap[index],
];
index = smallest;
}

isEmpty() {
  return this.heap.length === 0;
}
}

```

### Explanation:

1. **ListNode Class:** This represents the nodes of the linked lists.
2. **MinHeap Class:** Implements a min-heap to efficiently extract the smallest element.
3. **mergeKLists:** Main function that uses the heap to merge the sorted lists.

### Merging Multiple Sorted Data Sets (Optimized)

**547. Problem:** Given an array of k sorted linked lists, merge them into a single sorted linked list using a heap (priority queue). You need to efficiently implement this solution.

### Example:

#### Input:

```
lists = [
  [1, 4, 5],
  [1, 3, 4],
  [2, 6],
];
```

### Output:

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

### Solution:

```
class ListNode {
  constructor(val = 0, next = null) {
    this.val = val;
    this.next = next;
  }
}

var mergeKLists = function (lists) {
  const minHeap = new MinHeap();
  const resultHead = new ListNode();
  let current = resultHead;

  // Insert the first node of each list into the heap
  for (let i = 0; i < lists.length; i++) {
    if (lists[i]) {
      minHeap.insert(lists[i]);
    }
  }

  // Extract the smallest element and add the next node from the same list
  while (!minHeap.isEmpty()) {
    const node = minHeap.extractMin();
    current.next = node;
    current = current.next;

    if (node.next) {
      minHeap.insert(node.next);
    }
  }

  return resultHead.next;
};

// MinHeap class to support heap operations
```

```

class MinHeap {
  constructor() {
    this.heap = [];
  }

  insert(node) {
    this.heap.push(node);
    this.heapifyUp();
  }

  extractMin() {
    if (this.isEmpty()) return null;
    const minNode = this.heap[0];
    const lastNode = this.heap.pop();
    if (this.heap.length > 0) {
      this.heap[0] = lastNode;
      this.heapifyDown();
    }
    return minNode;
  }

  heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
      const parentIndex = Math.floor((index - 1) / 2);
      if (this.heap[index].val >= this.heap[parentIndex].val) break;
      [this.heap[index], this.heap[parentIndex]] = [
        this.heap[parentIndex],
        this.heap[index],
      ];
      index = parentIndex;
    }
  }

  heapifyDown() {
    let index = 0;
    const size = this.heap.length;
    while (index < size) {
      const leftChildIndex = 2 * index + 1;
      const rightChildIndex = 2 * index + 2;
      let smallest = index;

      if (
        leftChildIndex < size &&
        this.heap[leftChildIndex].val < this.heap[smallest].val
      ) {
        smallest = leftChildIndex;
      }
    }
  }
}

```

```

if (
    rightChildIndex < size &&
    this.heap[rightChildIndex].val < this.heap[smallest].val
) {
    smallest = rightChildIndex;
}

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [
    this.heap[smallest],
    this.heap[index],
];
index = smallest;
}
}

isEmpty() {
    return this.heap.length === 0;
}
}

```

### **Explanation:**

1. **Heap Initialization:**
  - o A MinHeap class is created to manage the heap operations. It allows us to insert, extract the minimum node, and perform heapify operations.
  - o We initialize the heap and insert the first node of each linked list into the heap.
2. **Heap Operations:**
  - o The heap stores the smallest element at the top, ensuring that we can always extract the smallest node efficiently.
  - o After extracting the node, if it has a next node, we insert that node into the heap.
3. **Result Construction:**
  - o As we extract the nodes, we link them together to form the merged sorted linked list.
4. **MinHeap Class:**
  - o It includes operations like insert, extractMin, heapifyUp, heapifyDown, and isEmpty to ensure the heap properties are maintained.

### **Finding the Top K Frequent Elements**

#### **548. Problem:**

**Given a non-empty array of integers, return the k most frequent elements. You need to implement a solution that efficiently computes this result using a heap.**

#### **Example:**

**Input:**

```
nums = [1, 1, 1, 2, 2, 3];
k = 2;
```

**Output:**

```
[1, 2]
```

**Solution:**

```
var topKFrequent = function (nums, k) {
    const frequencyMap = new Map();

    // Step 1: Count the frequency of each number
    for (let num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) || 0) + 1);
    }

    // Step 2: Use a min-heap to store the k most frequent elements
    const minHeap = new MinHeap();

    // Step 3: Insert each element and its frequency into the heap
    for (let [num, freq] of frequencyMap) {
        minHeap.insert([num, freq]);
        if (minHeap.size() > k) {
            minHeap.extractMin();
        }
    }

    // Step 4: Convert the heap to the result array
    const result = [];
    while (!minHeap.isEmpty()) {
        result.push(minHeap.extractMin()[0]);
    }

    return result;
};

// MinHeap class to support heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }

    // Insert an element into the heap
    insert(pair) {
        this.heap.push(pair);
        this.heapifyUp();
    }
}
```

```
// Extract the minimum element from the heap
extractMin() {
    if (this.isEmpty()) return null;
    const minElement = this.heap[0];
    const lastElement = this.heap.pop();
    if (this.heap.length > 0) {
        this.heap[0] = lastElement;
        this.heapifyDown();
    }
    return minElement;
}

// Check if the heap is empty
isEmpty() {
    return this.heap.length === 0;
}

// Get the size of the heap
size() {
    return this.heap.length;
}

// Heapify the heap upwards
heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
        const parentIndex = Math.floor((index - 1) / 2);
        if (this.heap[index][1] >= this.heap[parentIndex][1]) break;
        [this.heap[index], this.heap[parentIndex]] = [
            this.heap[parentIndex],
            this.heap[index],
        ];
        index = parentIndex;
    }
}

// Heapify the heap downwards
heapifyDown() {
    let index = 0;
    const size = this.heap.length;
    while (index < size) {
        const leftChildIndex = 2 * index + 1;
        const rightChildIndex = 2 * index + 2;
        let smallest = index;

        if (
            leftChildIndex < size &&
```

```

        this.heap[leftChildIndex][1] < this.heap[smallest][1]
    ) {
    smallest = leftChildIndex;
}

if (
    rightChildIndex < size &&
    this.heap[rightChildIndex][1] < this.heap[smallest][1]
) {
    smallest = rightChildIndex;
}

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [
    this.heap[smallest],
    this.heap[index],
];
index = smallest;
}
}
}

```

### **Explanation of the Code:**

- Frequency Counting:** We count how often each number appears in the array using a Map (frequencyMap).
- Min-Heap for Top K Elements:** We define a MinHeap class that stores pairs of [number, frequency]. The heap is maintained such that it always holds the k most frequent numbers.
- Insert and Maintain Heap Size:** We insert each number-frequency pair into the heap. If the heap grows beyond size k, we remove the least frequent element.
- Extracting the Result:** Once the heap is built, the Finding the Top K Frequent Elements are extracted, and the result is returned.

### **Finding the Top K Frequent Elements**

**549. Problem:** Given an integer array nums and an integer k, return the k most frequent elements from the array. You must solve the problem using a heap.

#### **Example:**

##### **Input:**

nums = [1, 1, 1, 2, 2, 3], (k = 2);

##### **Output:**

[1, 2]

**Solution:**

```
var topKFrequent = function (nums, k) {
  const frequencyMap = new Map();

  // Step 1: Count the frequency of each number
  for (let num of nums) {
    frequencyMap.set(num, (frequencyMap.get(num) || 0) + 1);
  }

  // Step 2: Use a min-heap to store the k most frequent elements
  const minHeap = new MinHeap();

  // Step 3: Insert each element and its frequency into the heap
  for (let [num, freq] of frequencyMap) {
    minHeap.insert([num, freq]);
    if (minHeap.size() > k) {
      minHeap.extractMin();
    }
  }

  // Step 4: Convert the heap to the result array
  const result = [];
  while (!minHeap.isEmpty()) {
    result.push(minHeap.extractMin()[0]);
  }

  return result;
};

// MinHeap class to support heap operations
class MinHeap {
  constructor() {
    this.heap = [];
  }

  // Insert an element into the heap
  insert(pair) {
    this.heap.push(pair);
    this.heapifyUp();
  }

  // Extract the minimum element from the heap
  extractMin() {
    if (this.isEmpty()) return null;
    const minElement = this.heap[0];
    const lastElement = this.heap.pop();
    if (this.heap.length > 0) {
      this.heap[0] = lastElement;
      this.heapifyDown();
    }
    return minElement;
  }

  // Helper function to heapify up
  heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
      const parentIndex = Math.floor((index - 1) / 2);
      if (this.heap[index][1] < this.heap[parentIndex][1]) {
        [this.heap[index], this.heap[parentIndex]] = [
          this.heap[parentIndex],
          this.heap[index]
        ];
        index = parentIndex;
      } else {
        break;
      }
    }
  }

  // Helper function to heapify down
  heapifyDown() {
    let index = 0;
    const length = this.heap.length;
    while (index < length) {
      const leftChildIndex = 2 * index + 1;
      const rightChildIndex = 2 * index + 2;
      let smallestIndex = index;
      if (leftChildIndex < length && this.heap[leftChildIndex][1] < this.heap[smallestIndex][1]) {
        smallestIndex = leftChildIndex;
      }
      if (rightChildIndex < length && this.heap[rightChildIndex][1] < this.heap[smallestIndex][1]) {
        smallestIndex = rightChildIndex;
      }
      if (smallestIndex === index) {
        break;
      }
      [this.heap[index], this.heap[smallestIndex]] = [
        this.heap[smallestIndex],
        this.heap[index]
      ];
      index = smallestIndex;
    }
  }

  isEmpty() {
    return this.heap.length === 0;
  }

  size() {
    return this.heap.length;
  }
}
```

```

        this.heapifyDown();
    }
    return minElement;
}

// Check if the heap is empty
isEmpty() {
    return this.heap.length === 0;
}

// Get the size of the heap
size() {
    return this.heap.length;
}

// Heapify the heap upwards
heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
        const parentIndex = Math.floor((index - 1) / 2);
        if (this.heap[index][1] > this.heap[parentIndex][1]) break;
        [this.heap[index], this.heap[parentIndex]] = [
            this.heap[parentIndex],
            this.heap[index],
        ];
        index = parentIndex;
    }
}

// Heapify the heap downwards
heapifyDown() {
    let index = 0;
    while (index < this.heap.length) {
        const leftChildIndex = 2 * index + 1;
        const rightChildIndex = 2 * index + 2;
        let smallest = index;

        if (
            leftChildIndex < this.heap.length &&
            this.heap[leftChildIndex][1] < this.heap[smallest][1]
        ) {
            smallest = leftChildIndex;
        }
        if (
            rightChildIndex < this.heap.length &&
            this.heap[rightChildIndex][1] < this.heap[smallest][1]
        ) {
            smallest = rightChildIndex;
        }
    }
}

```

```

    }
    if (smallest === index) break;
    [this.heap[index], this.heap[smallest]] = [
      this.heap[smallest],
      this.heap[index],
    ];
    index = smallest;
  }
}

```

### **Explanation of the Code:**

- **Counting Frequencies:** The frequencyMap stores the frequency of each element.
- **Min-Heap:** We use a custom MinHeap class to handle the heap operations. The heap stores pairs of [element, frequency] where the heap is ordered based on the frequency of the elements.
- **Inserting and Extracting from Heap:** We insert each element and its frequency into the heap. When the heap exceeds size k, we remove the element with the smallest frequency to ensure only the k most frequent elements remain.
- **Heap Operations:** The heapifyUp and heapifyDown methods ensure the heap property is maintained during insertions and removals.

This solution ensures that the k most frequent elements are efficiently computed using a min-heap.

### **Time Complexity:**

- Counting frequencies takes  $O(n)$ .
- Inserting into and removing from the heap takes  $O(\log k)$  for each element, resulting in a total time complexity of  $O(n \log k)$ .

### **Space Complexity:**

- $O(n)$  for the frequency map and  $O(k)$  for the heap.

## **Top K Frequent Words**

**550. Problem:** Given an array of strings words and an integer k, return the k most frequent words from the list. Sort the answer by frequency (from highest to lowest), and if two words have the same frequency, sort them in lexicographical order.

### **Example:**

#### **Input:**

```

words = ["i", "love", "leetcode", "i", "love", "coding"];
k = 2;

```

**Output:**

```
["i", "love"];
```

**Solution:**

```
var topKFrequent = function (words, k) {
    const frequencyMap = new Map();

    // Step 1: Count the frequency of each word
    for (let word of words) {
        frequencyMap.set(word, (frequencyMap.get(word) || 0) + 1);
    }

    // Step 2: Use a min-heap to store the k most frequent words
    const minHeap = new MinHeap();

    // Step 3: Insert each word and its frequency into the heap
    for (let [word, freq] of frequencyMap) {
        minHeap.insert([word, freq]);
        if (minHeap.size() > k) {
            minHeap.extractMin();
        }
    }

    // Step 4: Convert the heap to the result array
    const result = [];
    while (!minHeap.isEmpty()) {
        result.push(minHeap.extractMin()[0]);
    }

    // Reverse the result array to get the correct order (highest frequency first)
    return result.reverse();
};

// MinHeap class to support heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }

    // Insert an element into the heap
    insert(pair) {
        this.heap.push(pair);
        this.heapifyUp();
    }
}
```

```
// Extract the minimum element from the heap
extractMin() {
    if (this.isEmpty()) return null;
    const minElement = this.heap[0];
    const lastElement = this.heap.pop();
    if (this.heap.length > 0) {
        this.heap[0] = lastElement;
        this.heapifyDown();
    }
    return minElement;
}

// Check if the heap is empty
isEmpty() {
    return this.heap.length === 0;
}

// Get the size of the heap
size() {
    return this.heap.length;
}

// Heapify the heap upwards
heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
        const parentIndex = Math.floor((index - 1) / 2);
        if (
            this.heap[index][1] < this.heap[parentIndex][1] ||
            (this.heap[index][1] === this.heap[parentIndex][1] &&
             this.heap[index][0] < this.heap[parentIndex][0])
        ) {
            [this.heap[index], this.heap[parentIndex]] =
                [
                    this.heap[parentIndex],
                    this.heap[index],
                ];
            index = parentIndex;
        } else {
            break;
        }
    }
}

// Heapify the heap downwards
heapifyDown() {
    let index = 0;
    while (index * 2 + 1 < this.heap.length) {
```

```

const leftChildIndex = index * 2 + 1;
const rightChildIndex = index * 2 + 2;
let smallerChildIndex = leftChildIndex;

if (
  rightChildIndex < this.heap.length &&
  (this.heap[rightChildIndex][1] < this.heap[leftChildIndex][1] ||
   (this.heap[rightChildIndex][1] === this.heap[leftChildIndex][1] &&
    this.heap[rightChildIndex][0] < this.heap[leftChildIndex][0]))
) {
  smallerChildIndex = rightChildIndex;
}

if (
  this.heap[index][1] > this.heap[smallerChildIndex][1] ||
  (this.heap[index][1] === this.heap[smallerChildIndex][1] &&
   this.heap[index][0] > this.heap[smallerChildIndex][0])
) {
  [this.heap[index], this.heap[smallerChildIndex]] = [
    this.heap[smallerChildIndex],
    this.heap[index],
  ];
  index = smallerChildIndex;
} else {
  break;
}
}
}
}

```

### **Explanation:**

**Objective:** Find the k most frequent words in the list of words.

### **Steps:**

1. **Step 1:** Count the frequency of each word in the list.
2. **Step 2:** Use a min-heap (or priority queue) to store the words and their frequencies.
3. **Step 3:** Ensure that the heap contains only k elements at all times. The heap is sorted based on word frequency, and for words with equal frequency, it is sorted lexicographically.
4. **Step 4:** After processing all the words, extract the k most frequent words from the heap.

### **Why It Works:**

- The heap maintains the Finding the Top K Frequent Elements by removing the least frequent ones when the heap size exceeds k.

- Heap operations ensure that the insertion and removal of elements are efficient, with a time complexity of  $O(\log k)$  for each operation.

### Time Complexity:

- **Counting frequencies:**  $O(n)$  where  $n$  is the length of the array.
- **Heap operations:** For each element, we perform  $O(\log k)$  operations, leading to a total of  $O(n \log k)$  time complexity.

### Space Complexity:

- $O(n)$  for storing the frequency map, and  $O(k)$  for the heap.

## Finding the Top K Frequent Elements

**551. Problem:** Given a non-empty array of integers, return the  $k$  most frequent elements. You may assume that the answer is **guaranteed** to be unique.

### Example:

#### Input:

```
nums = [1, 1, 1, 2, 2, 3];
k = 2;
```

#### Output:

[1, 2]

### Solution:

```
var topKFrequent = function (nums, k) {
  const frequencyMap = new Map();

  // Step 1: Count the frequency of each element
  for (let num of nums) {
    frequencyMap.set(num, (frequencyMap.get(num) || 0) + 1);
  }

  // Step 2: Use a min-heap to store the Finding the Top K Frequent Elements
  const minHeap = new MinHeap();

  // Step 3: Insert each element and its frequency into the heap
  for (let [num, freq] of frequencyMap) {
    minHeap.insert([num, freq]);
    if (minHeap.size() > k) {
      minHeap.extractMin();
    }
  }
}
```

```

// Step 4: Extract the k most frequent elements from the heap
const result = [];
while (!minHeap.isEmpty()) {
    result.push(minHeap.extractMin()[0]);
}

return result;
};

// MinHeap class to support heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }

    // Insert an element into the heap
    insert(pair) {
        this.heap.push(pair);
        this.heapifyUp();
    }

    // Extract the minimum element from the heap
    extractMin() {
        if (this.isEmpty()) return null;
        const minElement = this.heap[0];
        const lastElement = this.heap.pop();
        if (this.heap.length > 0) {
            this.heap[0] = lastElement;
            this.heapifyDown();
        }
        return minElement;
    }

    // Check if the heap is empty
    isEmpty() {
        return this.heap.length === 0;
    }

    // Get the size of the heap
    size() {
        return this.heap.length;
    }

    // Heapify the heap upwards
    heapifyUp() {
        let index = this.heap.length - 1;
        while (index > 0) {

```

```

const parentIndex = Math.floor((index - 1) / 2);
if (this.heap[index][1] >= this.heap[parentIndex][1]) break;
[this.heap[index], this.heap[parentIndex]] = [
  this.heap[parentIndex],
  this.heap[index],
];
index = parentIndex;
}

// Heapify the heap downwards
heapifyDown() {
  let index = 0;
  const length = this.heap.length;
  while (index < length) {
    const leftChildIndex = 2 * index + 1;
    const rightChildIndex = 2 * index + 2;
    let smallest = index;

    if (
      leftChildIndex < length &&
      this.heap[leftChildIndex][1] < this.heap[smallest][1]
    ) {
      smallest = leftChildIndex;
    }
    if (
      rightChildIndex < length &&
      this.heap[rightChildIndex][1] < this.heap[smallest][1]
    ) {
      smallest = rightChildIndex;
    }
    if (smallest === index) break;
    [this.heap[index], this.heap[smallest]] = [
      this.heap[smallest],
      this.heap[index],
    ];
    index = smallest;
  }
}

```

### Explanation:

**Objective:** Find the k most frequent elements from the input list.

### Steps:

1. **Step 1:** Count the frequency of each element in the list using a hash map.
2. **Step 2:** Use a heap (min-heap) to store the Finding the Top K Frequent Elements. The heap will be used to efficiently find the k most frequent elements.
3. **Step 3:** Insert each element into the heap, and if the size of the heap exceeds k, remove the least frequent element (this ensures that we always keep the Finding the Top K Frequent Elements).
4. **Step 4:** Extract the elements from the heap to get the Finding the Top K Frequent Elements.

### Why It Works:

- The heap allows us to efficiently keep track of the Finding the Top K Frequent Elements. The operations on the heap (insertion and removal) have a time complexity of  $O(\log k)$ , making the approach efficient.

### Time Complexity:

- **Counting frequencies:**  $O(n)$  where n is the length of the array.
- **Heap operations:** For each element, we perform  $O(\log k)$  heap operations, leading to a total of  $O(n \log k)$  time complexity.

### Space Complexity:

- $O(n)$  for storing the frequency map, and  $O(k)$  for the heap.

## Finding the Top K Frequent Elements in a List

**552. Problem:** Given a non-empty list of words, return the top k frequent words in the list. The answer should be sorted by frequency from highest to lowest. If two words have the same frequency, they should be sorted alphabetically in ascending order.

### Example:

#### Input:

```
words = ["i", "love", "leetcode", "i", "love", "coding"];
k = 2;
```

#### Output:

```
["i", "love"]
```

#### Solution:

```
var topKFrequent = function (words, k) {
  // Step 1: Count frequency of each word
  const frequencyMap = new Map();
  for (let word of words) {
    frequencyMap.set(word, (frequencyMap.get(word) || 0) + 1);
  }
  // Step 2: Use a min-heap to store the top k frequent words
  const minHeap = new MinHeap();
  for (let [word, freq] of frequencyMap.entries()) {
    minHeap.insert({ freq, word });
    if (minHeap.size() > k) {
      minHeap.remove();
    }
  }
  // Step 3: Extract the words from the heap
  const result = [];
  while (!minHeap.isEmpty()) {
    result.push(minHeap.remove().word);
  }
  return result;
};
```

```

}

// Step 2: Use a min-heap to store Finding the Top K Frequent Elements
const minHeap = new MinHeap();

for (let [word, freq] of frequencyMap) {
    minHeap.insert([word, freq]);
    if (minHeap.size() > k) {
        minHeap.extractMin();
    }
}

// Step 3: Extract words from the heap and return the result
const result = [];
while (!minHeap.isEmpty()) {
    result.push(minHeap.extractMin()[0]);
}

// Since we used a min-heap, reverse the result to get top k words in correct order
return result.reverse();
};

// Min-Heap Class to support heap operations
class MinHeap {
    constructor() {
        this.heap = [];
    }

    insert(pair) {
        this.heap.push(pair);
        this.heapifyUp();
    }

    extractMin() {
        if (this.isEmpty()) return null;
        const minElement = this.heap[0];
        const lastElement = this.heap.pop();
        if (this.heap.length > 0) {
            this.heap[0] = lastElement;
            this.heapifyDown();
        }
        return minElement;
    }

    isEmpty() {
        return this.heap.length === 0;
    }
}

```

```

size() {
    return this.heap.length;
}

heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
        const parentIndex = Math.floor((index - 1) / 2);
        if (
            this.heap[index][1] > this.heap[parentIndex][1] ||
            (this.heap[index][1] === this.heap[parentIndex][1] &&
             this.heap[index][0] < this.heap[parentIndex][0])
        ) {
            break;
        }
        [this.heap[index], this.heap[parentIndex]] = [
            this.heap[parentIndex],
            this.heap[index],
        ];
        index = parentIndex;
    }
}

heapifyDown() {
    let index = 0;
    while (index < this.heap.length) {
        const leftChild = 2 * index + 1;
        const rightChild = 2 * index + 2;
        let smallest = index;

        if (
            leftChild < this.heap.length &&
            (this.heap[leftChild][1] < this.heap[smallest][1] ||
             (this.heap[leftChild][1] === this.heap[smallest][1] &&
              this.heap[leftChild][0] < this.heap[smallest][0]))
        ) {
            smallest = leftChild;
        }

        if (
            rightChild < this.heap.length &&
            (this.heap[rightChild][1] < this.heap[smallest][1] ||
             (this.heap[rightChild][1] === this.heap[smallest][1] &&
              this.heap[rightChild][0] < this.heap[smallest][0]))
        ) {
            smallest = rightChild;
        }
    }
}

```

```

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [
  this.heap[smallest],
  this.heap[index],
];
index = smallest;
}
}
}

```

### **Explanation:**

**Objective:** Find the k most frequent words, sorting them by frequency and alphabetically in case of ties.

### **Steps:**

1. **Step 1:** Use a hash map to count the frequency of each word.
2. **Step 2:** Insert each word along with its frequency into a heap (min-heap). If the heap exceeds size k, remove the least frequent word.
3. **Step 3:** Extract the words from the heap, and sort them by frequency and lexicographically.

### **Why It Works:**

- Using a heap allows us to efficiently keep track of the top k frequent words. Each heap operation (insertion and extraction) is  $O(\log k)$ , which ensures that we maintain the correct order.

### **Time Complexity:**

- **Counting Frequencies:**  $O(n)$  where n is the number of words.
- **Heap Operations:** Inserting into the heap takes  $O(\log k)$  time. Since we have n words, this step takes  $O(n \log k)$ .
- **Sorting the Heap:** Extracting the words from the heap takes  $O(k \log k)$  time.

**Space Complexity:**  $O(n)$  for the frequency map and  $O(k)$  for the heap.

## **Handling Streaming Data for Median Computation**

### **553. Problem:**

Design a data structure that supports the following two operations:

1. **Insert number:** Adds a number to the stream of numbers.
2. **Find median:** Returns the median of all numbers in the stream.

The **median** is the middle element in a sorted list of numbers. If the list has an even number of elements, there are two middle numbers and the median is the average of those two numbers.

### Example 1:

```
Insert(1)
Insert(2)
findMedian() -> 1.5
Insert(3)
findMedian() -> 2
```

### Explanation:

1. After inserting 1 and 2, the list is [1, 2]. The median is  $(1 + 2) / 2 = 1.5$ .
2. After inserting 3, the list is [1, 2, 3]. The median is 2.

### Solution:

```
class MedianFinder {
    constructor() {
        this.maxHeap = new MaxHeap();
        this.minHeap = new MinHeap();
    }

    insertNum(num) {
        // Step 1: Insert into the appropriate heap
        if (this.maxHeap.size() === 0 || num <= this.maxHeap.peek()) {
            this.maxHeap.insert(num);
        } else {
            this.minHeap.insert(num);
        }

        // Step 2: Balance the heaps
        if (this.maxHeap.size() > this.minHeap.size() + 1) {
            this.minHeap.insert(this.maxHeap.extractMax());
        } else if (this.minHeap.size() > this.maxHeap.size()) {
            this.maxHeap.insert(this.minHeap.extractMin());
        }
    }

    findMedian() {
        if (this.maxHeap.size() > this.minHeap.size()) {
            return this.maxHeap.peek();
        } else {
            return (this.maxHeap.peek() + this.minHeap.peek()) / 2;
        }
    }
}
```

```
}
```

```
// Max-Heap class
```

```
class MaxHeap {
```

```
    constructor() {
```

```
        this.heap = [];
    }
```

```

    insert(val) {
        this.heap.push(val);
        this.heapifyUp();
    }
```

```

    extractMax() {
        if (this.isEmpty()) return null;
        const max = this.heap[0];
        const last = this.heap.pop();
        if (this.heap.length > 0) {
            this.heap[0] = last;
            this.heapifyDown();
        }
        return max;
    }
```

```

    peek() {
        return this.isEmpty() ? null : this.heap[0];
    }
```

```

    size() {
        return this.heap.length;
    }
```

```

    isEmpty() {
        return this.heap.length === 0;
    }
```

```

    heapifyUp() {
        let index = this.heap.length - 1;
        while (index > 0) {
            let parentIndex = Math.floor((index - 1) / 2);
            if (this.heap[index] <= this.heap[parentIndex]) break;
            [this.heap[index], this.heap[parentIndex]] = [
                this.heap[parentIndex],
                this.heap[index],
            ];
            index = parentIndex;
        }
    }
```

```

heapifyDown() {
    let index = 0;
    while (2 * index + 1 < this.heap.length) {
        let leftChild = 2 * index + 1;
        let rightChild = 2 * index + 2;
        let largest = index;

        if (
            leftChild < this.heap.length &&
            this.heap[leftChild] > this.heap[largest]
        ) {
            largest = leftChild;
        }
        if (
            rightChild < this.heap.length &&
            this.heap[rightChild] > this.heap[largest]
        ) {
            largest = rightChild;
        }
        if (largest === index) break;
        [this.heap[index], this.heap[largest]] = [
            this.heap[largest],
            this.heap[index],
        ];
        index = largest;
    }
}

// Min-Heap class
class MinHeap {
    constructor() {
        this.heap = [];
    }

    insert(val) {
        this.heap.push(val);
        this.heapifyUp();
    }

    extractMin() {
        if (this.isEmpty()) return null;
        const min = this.heap[0];
        const last = this.heap.pop();
        if (this.heap.length > 0) {
            this.heap[0] = last;
            this.heapifyDown();
        }
        return min;
    }
}

```

```
        }
        return min;
    }

    peek() {
        return this.isEmpty() ? null : this.heap[0];
    }

    size() {
        return this.heap.length;
    }

    isEmpty() {
        return this.heap.length === 0;
    }

    heapifyUp() {
        let index = this.heap.length - 1;
        while (index > 0) {
            let parentIndex = Math.floor((index - 1) / 2);
            if (this.heap[index] >= this.heap[parentIndex]) break;
            [this.heap[index], this.heap[parentIndex]] = [
                this.heap[parentIndex],
                this.heap[index],
            ];
            index = parentIndex;
        }
    }

    heapifyDown() {
        let index = 0;
        while (2 * index + 1 < this.heap.length) {
            let leftChild = 2 * index + 1;
            let rightChild = 2 * index + 2;
            let smallest = index;

            if (
                leftChild < this.heap.length &&
                this.heap[leftChild] < this.heap[smallest]
            ) {
                smallest = leftChild;
            }
            if (
                rightChild < this.heap.length &&
                this.heap[rightChild] < this.heap[smallest]
            ) {
                smallest = rightChild;
            }
        }
    }
}
```

```

    if (smallest === index) break;
    [this.heap[index], this.heap[smallest]] = [
      this.heap[smallest],
      this.heap[index],
    ];
    index = smallest;
  }
}

// Usage
const finder = new MedianFinder();
finder.insertNum(1);
finder.insertNum(2);
console.log(finder.findMedian()); // Output: 1.5
finder.insertNum(3);
console.log(finder.findMedian()); // Output: 2

```

### Explanation of the Code:

- **MedianFinder:**
  - This class maintains two heaps: maxHeap (for the smaller half of the numbers) and minHeap (for the larger half).
  - The insertNum function inserts a new number into the appropriate heap and ensures the heaps are balanced.
  - The findMedian function returns the median, which is either the top of the maxHeap if the heaps are of unequal sizes or the average of the tops of both heaps if the sizes are equal.
- **MaxHeap and MinHeap:**
  - These are custom heap classes implemented to manage the priority of elements. The maxHeap always returns the largest element (root), and the minHeap always returns the smallest element.

## Handling Streaming Data for Median Computation (Heap Approach)

### 554. Problem:

Design a data structure that supports the following two operations:

1. **Insert number:** Adds a number to the stream of numbers.
2. **Find median:** Returns the median of all numbers in the stream.

The **median** is the middle element in a sorted list of numbers. If the list has an even number of elements, the median is the average of the two middle elements.

You need to implement these operations efficiently for large data streams.

### Example 1:

```

Insert(1)
Insert(2)
findMedian() -> 1.5
Insert(3)
findMedian() -> 2

```

### Explanation:

1. After inserting 1 and 2, the list is [1, 2]. The median is  $(1 + 2) / 2 = 1.5$ .
2. After inserting 3, the list is [1, 2, 3]. The median is 2.

### Solution:

```

class MedianFinder {
    constructor() {
        this.maxHeap = new MaxHeap();
        this.minHeap = new MinHeap();
    }

    insertNum(num) {
        // Step 1: Insert the number into the appropriate heap
        if (this.maxHeap.size() === 0 || num <= this.maxHeap.peek()) {
            this.maxHeap.insert(num);
        } else {
            this.minHeap.insert(num);
        }

        // Step 2: Balance the heaps
        if (this.maxHeap.size() > this.minHeap.size() + 1) {
            this.minHeap.insert(this.maxHeap.extractMax());
        } else if (this.minHeap.size() > this.maxHeap.size()) {
            this.maxHeap.insert(this.minHeap.extractMin());
        }
    }

    findMedian() {
        if (this.maxHeap.size() > this.minHeap.size()) {
            return this.maxHeap.peek();
        } else {
            return (this.maxHeap.peek() + this.minHeap.peek()) / 2;
        }
    }
}

// Max-Heap class
class MaxHeap {
    constructor() {
        this.heap = [];
    }
}

```

```
}

insert(val) {
    this.heap.push(val);
    this.heapifyUp();
}

extractMax() {
    if (this.isEmpty()) return null;
    const max = this.heap[0];
    const last = this.heap.pop();
    if (this.heap.length > 0) {
        this.heap[0] = last;
        this.heapifyDown();
    }
    return max;
}

peek() {
    return this.heap[0];
}

size() {
    return this.heap.length;
}

isEmpty() {
    return this.heap.length === 0;
}

heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
        const parentIndex = Math.floor((index - 1) / 2);
        if (this.heap[parentIndex] >= this.heap[index]) break;
        [this.heap[parentIndex], this.heap[index]] = [
            this.heap[index],
            this.heap[parentIndex],
        ];
        index = parentIndex;
    }
}

heapifyDown() {
    let index = 0;
    while (index < this.heap.length) {
        const leftChild = 2 * index + 1;
        const rightChild = 2 * index + 2;
```

```

let largest = index;

if (
  leftChild < this.heap.length &&
  this.heap[leftChild] > this.heap[largest]
) {
  largest = leftChild;
}

if (
  rightChild < this.heap.length &&
  this.heap[rightChild] > this.heap[largest]
) {
  largest = rightChild;
}

if (largest === index) break;

[this.heap[index], this.heap[largest]] = [
  this.heap[largest],
  this.heap[index],
];
index = largest;
}

}

// Min-Heap class
class MinHeap {
  constructor() {
    this.heap = [];
  }

  insert(val) {
    this.heap.push(val);
    this.heapifyUp();
  }

  extractMin() {
    if (this.isEmpty()) return null;
    const min = this.heap[0];
    const last = this.heap.pop();
    if (this.heap.length > 0) {
      this.heap[0] = last;
      this.heapifyDown();
    }
    return min;
  }
}

```

```
peek() {
    return this.heap[0];
}

size() {
    return this.heap.length;
}

isEmpty() {
    return this.heap.length === 0;
}

heapifyUp() {
    let index = this.heap.length - 1;
    while (index > 0) {
        const parentIndex = Math.floor((index - 1) / 2);
        if (this.heap[parentIndex] <= this.heap[index]) break;
        [this.heap[parentIndex], this.heap[index]] = [
            this.heap[index],
            this.heap[parentIndex],
        ];
        index = parentIndex;
    }
}

heapifyDown() {
    let index = 0;
    while (index < this.heap.length) {
        const leftChild = 2 * index + 1;
        const rightChild = 2 * index + 2;
        let smallest = index;

        if (
            leftChild < this.heap.length &&
            this.heap[leftChild] < this.heap[smallest]
        ) {
            smallest = leftChild;
        }

        if (
            rightChild < this.heap.length &&
            this.heap[rightChild] < this.heap[smallest]
        ) {
            smallest = rightChild;
        }

        if (smallest === index) break;
        [this.heap[index], this.heap[smallest]] = [
            this.heap[smallest],
            this.heap[index],
        ];
        index = smallest;
    }
}
```

```

        [this.heap[index], this.heap[smallest]] =
            [this.heap[smallest],
             this.heap[index],
            ];
        index = smallest;
    }
}
}

// Example usage:
const medianFinder = new MedianFinder();
medianFinder.insertNum(1);
medianFinder.insertNum(2);
console.log(medianFinder.findMedian()); // 1.5
medianFinder.insertNum(3);
console.log(medianFinder.findMedian()); // 2

```

## Explanation:

1. **Two heaps approach:**
  - o **Max-Heap:** This heap stores the smaller half of the numbers in the stream. The root of the max-heap represents the largest number of the smaller half.
  - o **Min-Heap:** This heap stores the larger half of the numbers in the stream. The root of the min-heap represents the smallest number of the larger half.
2. **Insert Operation:**
  - o Insert each number into the appropriate heap:
    - If the number is smaller than or equal to the maximum element in the max-heap, insert it into the max-heap.
    - Otherwise, insert it into the min-heap.
  - o **Balance the heaps:** Ensure the heaps are balanced. The number of elements in both heaps should differ by at most 1. If one heap has more than one extra element, transfer the root element from that heap to the other heap.
3. **Find Median:**
  - o If the number of elements is odd, the median will be the root of the heap with more elements.
  - o If the number of elements is even, the median will be the average of the roots of the two heaps.

## Why it Works:

- The max-heap stores the smaller half of the numbers, and the min-heap stores the larger half. By maintaining the heaps in balance, we can quickly access the middle element(s) to compute the median.

## Time Complexity:

- **Insert Operation:**  $O(\log n)$  — because we are inserting into a heap and the heapify operation takes logarithmic time.

- **Find Median Operation:**  $O(1)$  — because we only need to access the roots of the heaps.

### Space Complexity:

- $O(n)$  where  $n$  is the number of elements in the stream, as we are storing all elements in the two heaps.

## Handling Streaming Data for Median Computation (Advanced Version)

### 555. Problem:

Implement a data structure that supports the following two operations:

1. **Insert number:** Adds a number to the data stream.
2. **Find median:** Returns the median of all numbers inserted so far.

You need to implement the data structure efficiently to support both operations.

### Example 1:

```
insert(1)
insert(2)
findMedian() -> 1.5
insert(3)
findMedian() -> 2
```

### Explanation:

1. After inserting 1 and 2, the numbers are [1, 2]. The median is  $(1 + 2) / 2 = 1.5$ .
2. After inserting 3, the numbers are [1, 2, 3]. The median is 2.

### Solution:

```
class MedianFinder {
    constructor() {
        this.maxHeap = new MaxHeap();
        this.minHeap = new MinHeap();
    }

    insertNum(num) {
        if (this.maxHeap.size() === 0 || num <= this.maxHeap.peek()) {
            this.maxHeap.insert(num);
        } else {
            this.minHeap.insert(num);
        }

        if (this.maxHeap.size() > this.minHeap.size() + 1) {
            this.minHeap.insert(this.maxHeap.extractMax());
        }
    }
}
```

```

} else if (this.minHeap.size() > this.maxHeap.size()) {
    this.maxHeap.insert(this.minHeap.extractMin());
}
}

findMedian() {
    if (this.maxHeap.size() > this.minHeap.size()) {
        return this.maxHeap.peek();
    } else {
        return (this.maxHeap.peek() + this.minHeap.peek()) / 2;
    }
}
}

class MaxHeap {
    constructor() {
        this.heap = [];
    }

    insert(val) {
        this.heap.push(val);
        this.heapifyUp();
    }

    extractMax() {
        if (this.isEmpty()) return null;
        const max = this.heap[0];
        const last = this.heap.pop();
        if (this.heap.length > 0) {
            this.heap[0] = last;
            this.heapifyDown();
        }
        return max;
    }

    peek() {
        return this.heap[0];
    }

    size() {
        return this.heap.length;
    }

    isEmpty() {
        return this.heap.length === 0;
    }

    heapifyUp() {

```

```

let index = this.heap.length - 1;
while (index > 0) {
  const parent = Math.floor((index - 1) / 2);
  if (this.heap[index] > this.heap[parent]) {
    [this.heap[index], this.heap[parent]] = [
      this.heap[parent],
      this.heap[index],
    ];
    index = parent;
  } else {
    break;
  }
}

heapifyDown() {
  let index = 0;
  const length = this.heap.length;
  while (index < length) {
    let leftChild = 2 * index + 1;
    let rightChild = 2 * index + 2;
    let largest = index;

    if (leftChild < length && this.heap[leftChild] > this.heap[largest]) {
      largest = leftChild;
    }
    if (rightChild < length && this.heap[rightChild] > this.heap[largest]) {
      largest = rightChild;
    }
    if (largest !== index) {
      [this.heap[index], this.heap[largest]] = [
        this.heap[largest],
        this.heap[index],
      ];
      index = largest;
    } else {
      break;
    }
  }
}

class MinHeap {
  constructor() {
    this.heap = [];
  }

  insert(val) {

```

```
this.heap.push(val);
this.heapifyUp();
}

extractMin() {
  if (this.isEmpty()) return null;
  const min = this.heap[0];
  const last = this.heap.pop();
  if (this.heap.length > 0) {
    this.heap[0] = last;
    this.heapifyDown();
  }
  return min;
}

peek() {
  return this.heap[0];
}

size() {
  return this.heap.length;
}

isEmpty() {
  return this.heap.length === 0;
}

heapifyUp() {
  let index = this.heap.length - 1;
  while (index > 0) {
    const parent = Math.floor((index - 1) / 2);
    if (this.heap[index] < this.heap[parent]) {
      [this.heap[index], this.heap[parent]] = [
        this.heap[parent],
        this.heap[index],
      ];
      index = parent;
    } else {
      break;
    }
  }
}

heapifyDown() {
  let index = 0;
  const length = this.heap.length;
  while (index < length) {
    let leftChild = 2 * index + 1;
```

```

let rightChild = 2 * index + 2;
let smallest = index;

if (leftChild < length && this.heap[leftChild] < this.heap[smallest]) {
    smallest = leftChild;
}
if (rightChild < length && this.heap[rightChild] < this.heap[smallest]) {
    smallest = rightChild;
}
if (smallest !== index) {
    [this.heap[index], this.heap[smallest]] = [
        this.heap[smallest],
        this.heap[index],
    ];
    index = smallest;
} else {
    break;
}
}
}
}

```

### Explanation:

1. **Two Heaps:**
  - o Use a **Max-Heap** to store the smaller half of the numbers.
  - o Use a **Min-Heap** to store the larger half of the numbers.
2. **Insert Operation:**
  - o If the number is less than or equal to the maximum of the Max-Heap, insert it into the Max-Heap.
  - o Otherwise, insert it into the Min-Heap.
3. **Balance the Heaps:**
  - o After each insertion, make sure the two heaps are balanced in terms of size. The Max-Heap can only have at most one extra element than the Min-Heap.
  - o If the size difference becomes greater than 1, transfer the root of the heap with more elements to the other heap.
4. **Find Median:**
  - o If the two heaps are of equal size, the median is the average of the roots of both heaps.
  - o If the Max-Heap has one extra element, the median is the root of the Max-Heap.

### Time Complexity:

- **Insert** operation:  $O(\log n)$  for the insertion into a heap and for balancing.
- **Find Median** operation:  $O(1)$  because it only requires accessing the root of the heaps.

### Space Complexity:

- $O(n)$  for storing the elements in the heaps.

## 10. Graphs & Connectivity Problems

### Cloning Graph Structures

#### 556. Problem:

Given a reference to a node in a **connected undirected graph**, return a deep copy (clone) of the graph. Each node in the graph contains a value and a list of its neighbors.

Implement a solution to clone the graph. You need to clone the graph and return a reference to the cloned graph.

Each node in the graph has the following fields:

- `val`: an integer representing the node's value.
- `neighbors`: a list of the node's neighbors.

#### Example:

#### Input:

```
Graph = [
  [2, 4],
  [1, 3],
  [2, 4],
  [1, 3],
];
```

#### Output:

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

#### Solution:

```
/** 
 * // Definition for a Node.
 * function Node(val, neighbors) {
 *   this.val = val;
 *   this.neighbors = neighbors || [];
```

```

* };
*/
/**
* @param {Node} node
* @return {Node}
*/
var cloneGraph = function (node) {
  if (!node) return null;

  const map = new Map();

  const dfs = (n) => {
    if (map.has(n)) {
      return map.get(n);
    }

    // Create a new node with the same value as the original node
    const clone = new Node(n.val);
    map.set(n, clone);

    // Recursively clone all the neighbors
    for (let neighbor of n.neighbors) {
      clone.neighbors.push(dfs(neighbor));
    }
  }

  return clone;
};

return dfs(node);
};

```

### Explanation:

#### 1. Node Constructor:

- We first define the Node class, which represents a node in the graph, with a value val and an array neighbors.

#### 2. Base Case:

- If the input node is null, return null.

#### 3. DFS Helper Function:

- The dfs function is a recursive helper function that clones a node and its neighbors.
- If a node has already been cloned (i.e., it exists in the map), we simply return the cloned node to prevent cycles.
- If the node has not been cloned, we create a new node with the same value and then recursively clone all of its neighbors.

#### 4. Hash Map:

- The map stores the mapping between original nodes and their corresponding clones. This ensures that if we encounter a node again (due to cycles or multiple neighbors), we reuse the already cloned node.

#### 5. Final Return:

- After initiating the DFS traversal from the given node, the final result will be the root of the cloned graph.

#### Time Complexity:

- **O(N)** where N is the number of nodes in the graph. Each node is visited once during the DFS traversal.

#### Space Complexity:

- **O(N)** where N is the number of nodes. The space complexity arises from the recursion stack and the space used by the hash map to store the cloned nodes.

### Planning Task Dependencies

**557. Problem:** There are a total of n courses you have to take, labeled from 0 to n - 1. You are given an array of prerequisite pairs prerequisites where prerequisites[i] = [a, b] indicates that to take course a, you have to first take course b. Return true if you can finish all courses. Otherwise, return false.

#### Example:

#### Input:

```
(n = 2), (prerequisites = [[1, 0]]);
```

#### Output:

True

Explanation: There are two courses, 0 and 1, and the only prerequisite is that course 1 has to be taken after course 0. You can finish all courses by taking the courses in order: [0, 1].

#### Solution:

```
/**
 * @param {number} numCourses
 * @param {number[][]} prerequisites
 * @return {boolean}
 */
var canFinish = function (numCourses, prerequisites) {
  // Create an adjacency list to represent the graph
  const adjList = new Array(numCourses).fill().map(() => []);
  const indegree = new Array(numCourses).fill(0);
  ...
```

```

// Build the graph and calculate the indegrees
for (let [course, pre] of prerequisites) {
    adjList[pre].push(course);
    indegree[course]++;
}

// Initialize a queue with courses having no prerequisites
const queue = [];
for (let i = 0; i < numCourses; i++) {
    if (indegree[i] === 0) {
        queue.push(i);
    }
}

// Process the courses from the queue
let processedCourses = 0;
while (queue.length > 0) {
    const course = queue.shift();
    processedCourses++;

    for (let nextCourse of adjList[course]) {
        indegree[nextCourse]--;
        if (indegree[nextCourse] === 0) {
            queue.push(nextCourse);
        }
    }
}

// If we processed all courses, it means there is no cycle
return processedCourses === numCourses;
};

```

### **Explanation:**

#### **1. Adjacency List:**

- We first create an adjacency list `adjList` where each index represents a course, and each element at that index is a list of courses that depend on it.

#### **2. Indegree Array:**

- The indegree array tracks the number of prerequisites for each course. Initially, all courses have an indegree of 0, but for each prerequisite pair, we increment the indegree of the dependent course.

#### **3. Queue Initialization:**

- We initialize a queue with all the courses that have no prerequisites (`indegree[i] === 0`).

#### **4. BFS / Kahn's Algorithm:**

- We process each course from the queue. For each course, we decrement the indegree of its dependent courses. If any course's indegree becomes 0, it is added to the queue.

#### 5. Cycle Detection:

- If the total number of processed courses equals numCourses, it means there is no cycle in the graph, and we can finish all courses.
- If the number of processed courses is less than numCourses, it indicates a cycle exists, and it's impossible to finish all courses.

#### Time Complexity:

- $O(V + E)$ , where V is the number of courses and E is the number of prerequisite pairs. This is because we traverse each course and its dependencies once.

#### Space Complexity:

- $O(V + E)$ , for the adjacency list and the indegree array, where V is the number of courses and E is the number of prerequisite pairs.

### Tracking Water Flow Across Networks

#### 558. Problem:

Given an  $m \times n$  matrix of non-negative integers representing the height of each cell in a grid, the "Pacific Ocean" touches the left and top edges of the grid and the "Atlantic Ocean" touches the right and bottom edges of the grid.

Water can flow from a cell to another cell if the height of the destination cell is less than or equal to the height of the source cell. For example, if a cell has a height of 5, it can flow to a neighboring cell with a height of 3 or 4, but not 6.

Return a list of coordinates (i.e.,  $(x, y)$ ) where water can flow to both the Pacific and Atlantic oceans.

#### Example:

#### Input:

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

#### Output:

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

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

### Solution:

```
/**  
 * @param {number[][]} matrix  
 * @return {number[][]}  
 */  
var pacificAtlantic = function (matrix) {  
    if (matrix.length === 0 || matrix[0].length === 0) return [];  
  
    const m = matrix.length,  
        n = matrix[0].length;  
    const pacific = Array.from({ length: m }, () => Array(n).fill(false));  
    const atlantic = Array.from({ length: m }, () => Array(n).fill(false));  
  
    // Helper function for DFS  
    const dfs = (i, j, visited, prevHeight) => {  
        if (  
            i < 0 ||  
            i >= m ||  
            j < 0 ||  
            j >= n ||  
            visited[i][j] ||  
            matrix[i][j] < prevHeight  
        ) {  
            return;  
        }  
        visited[i][j] = true;  
        // Directions: up, down, left, right  
        const directions = [  
            [-1, 0],  
            [1, 0],  
            [0, -1],  
            [0, 1],  
        ];  
        for (const [di, dj] of directions) {  
            dfs(i + di, j + dj, visited, matrix[i][j]);  
        }  
    };  
  
    // Perform DFS for Pacific ocean (top and left edges)  
    for (let i = 0; i < m; i++) {  
        dfs(i, 0, pacific, matrix[i][0]);  
    }  
    for (let j = 0; j < n; j++) {  
        dfs(0, j, pacific, matrix[0][j]);  
    }  
};
```

```

// Perform DFS for Atlantic ocean (bottom and right edges)
for (let i = 0; i < m; i++) {
    dfs(i, n - 1, atlantic, matrix[i][n - 1]);
}
for (let j = 0; j < n; j++) {
    dfs(m - 1, j, atlantic, matrix[m - 1][j]);
}

// Find the intersection of cells that can reach both oceans
const result = [];
for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
        if (pacific[i][j] && atlantic[i][j]) {
            result.push([i, j]);
        }
    }
}

return result;
};

```

### **Explanation:**

#### **1. DFS Traversal:**

- A helper function `dfs(i, j, visited, prevHeight)` is defined to perform the DFS. This function:
  - Marks the current cell  $(i, j)$  as visited.
  - Explores the neighboring cells in all four directions (up, down, left, right) as long as the height of the neighboring cell is greater than or equal to the current cell's height.

#### **2. Starting DFS from Edges:**

- We perform DFS starting from the edges:
  - For the **Pacific Ocean**, we start from the top and left edges.
  - For the **Atlantic Ocean**, we start from the bottom and right edges.
- Each DFS will mark cells that can reach the respective ocean.

#### **3. Result:**

- After performing DFS for both oceans, we check each cell in the matrix. If a cell can reach both oceans (i.e., it's marked as true in both `pacific` and `atlantic` arrays), we add it to the result.

### **Time and Space Complexity:**

- **Time Complexity:**  $O(m * n)$ , where  $m$  is the number of rows and  $n$  is the number of columns. We perform a DFS from every cell in the matrix once.
- **Space Complexity:**  $O(m * n)$  for the `pacific`, `atlantic`, and recursion stack.

### **Finding the Longest Connected Sequences**

**559. Problem: Given an unsorted array of integers, find the length of the longest consecutive elements sequence.**

**Example:**

**Input:**

```
nums = [100, 4, 200, 1, 3, 2];
```

**Output:**

4

**Explanation:** The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore, the answer is 4.

**Constraints:**

- $0 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

**Solution:**

```
/**  
 * @param {number[]} nums  
 * @return {number}  
 */  
var longestConsecutive = function (nums) {  
    // Convert array to set for O(1) lookups  
    const set = new Set(nums);  
    let longest = 0;  
  
    for (let num of nums) {  
        // Only start a sequence if num-1 is not in the set  
        if (!set.has(num - 1)) {  
            let currentNum = num;  
            let currentStreak = 1;  
  
            // Expand the sequence by checking consecutive numbers  
            while (set.has(currentNum + 1)) {  
                currentNum++;  
                currentStreak++;  
            }  
  
            // Update the longest streak found  
            longest = Math.max(longest, currentStreak);  
        }  
    }  
};
```

```
    return longest;
};
```

### Explanation:

#### 1. Set Construction:

- A set is created from the nums array, allowing for constant-time lookups. This helps in checking whether a number is part of the sequence.

#### 2. Iterate Through Each Number:

- For each number, check if it is the beginning of a sequence by checking whether num - 1 is in the set. If it's not, we know that this number is the smallest in a possible sequence.

#### 3. Expand the Sequence:

- Starting from num, keep checking whether the next consecutive number (currentNum + 1) exists in the set. If it does, increment the streak counter.

#### 4. Track Maximum Streak:

- Keep track of the maximum sequence length found during the iterations.

### Time Complexity:

- The time complexity is **O(n)**, where n is the number of elements in the input array. This is because we loop through each element in the array, and each lookup or insertion into the set takes **O(1)** time.
- The space complexity is also **O(n)** due to the storage of the set.

## Reel 15:

### Pattern-Based Questions:

#### 1. Right-Angled Triangle Pattern

**Input:** N = 5

**Output:**

```
*
```

```
* *
```

```
* * *
```

```
* * * *
```

```
* * * * *
```

**Code:**

```
function trianglePattern(N) {  
    for (let i = 1; i <= N; i++) {  
        console.log("* ".repeat(i).trim());  
    }  
}  
trianglePattern(5);
```

**Explanation:**

##### 1. Outer Loop (i from 1 to N):

Each iteration represents a new row.

For i = 1, print 1 \*. For i = 2, print 2 \*, and so on.

##### 2. repeat() Method:

'\* '.repeat(i) generates a string with i stars followed by spaces.

##### 3. trim():

Removes the trailing space at the end of each row for a clean appearance.

##### 4. Output:

The number of stars in each row corresponds to the row number.

#### 2. Number Triangle

**Input:** N = 4

**Output:**

```
1  
2 3  
4 5 6
```

```
7 8 9 10
```

**Code:**

```
function numberTriangle(N) {  
    let num = 1;  
    for (let i = 1; i <= N; i++) {  
        let row = "";  
        for (let j = 1; j <= i; j++) {  
            row += num + " ";  
            num++;  
        }  
        console.log(row.trim());  
    }  
}  
numberTriangle(4);
```

**Explanation:**

**1. Outer Loop (i from 1 to N):**

Each iteration represents a new row.

**2. Inner Loop (j from 1 to i):**

Controls how many numbers are printed in the current row.

For  $i = 1$ , only 1 number is printed. For  $i = 2$ , 2 numbers are printed.

**3. Tracking Numbers:**

Start with  $num = 1$ .

Increment  $num$  after printing it in each column ( $j$ ).

**4. Output:**

The numbers in the triangle increment sequentially across rows.

**3. Pyramid Pattern**

**Input:**  $N = 4$

**Output:**

```
*
```

```
* *
```

```
* * *
```

```
* * * *
```

**Code:**

```
function pyramidPattern(N) {  
    for (let i = 1; i <= N; i++) {  
        let spaces = " ".repeat(N - i);  
        let stars = "* ".repeat(i);  
        console.log(spaces + stars.trim());  
    }  
}  
pyramidPattern(4);
```

**Explanation:****1. Outer Loop (i from 1 to N):**

Each iteration represents a row.

**2. Spaces:**

$N - i$  spaces are added before the stars to center-align them.

**3. Stars:**

'\* '.repeat(i) generates the required number of stars for the current row.

**4. Trim and Combine:**

Combine spaces and stars, then remove any trailing space with .trim().

**5. Output:**

The stars are center-aligned, forming a pyramid shape.

**4. Diamond Pattern****Input:** N = 3**Output:**

```
*  
* *  
* * *  
* *  
*
```

**Code:**

```
function diamondPattern(N) {  
    for (let i = 1; i <= N; i++) {  
        console.log(" ".repeat(N - i) + "* ".repeat(i).trim());  
    }  
}
```

```

}
for (let i = N - 1; i >= 1; i--) {
  console.log(" ".repeat(N - i) + "* ".repeat(i).trim());
}
}
diamondPattern(3);

```

### **Explanation:**

#### **1. Top Half:**

First loop (i from 1 to N) creates the top half of the diamond.

Add N - i spaces before stars to center-align them.

#### **2. Bottom Half:**

Second loop (i from N - 1 to 1) creates the bottom half of the diamond.

#### **3. Stars:**

Use '\* '.repeat(i) to generate stars for each row.

#### **4. Output:**

The diamond pattern is formed with a symmetric arrangement of stars.

### **5. Reverse Number Triangle**

**Input:** N = 5

**Output:**

```

1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

```

### **Code:**

```

function reverseNumberTriangle(N) {
  for (let i = N; i >= 1; i--) {
    let row = "";
    for (let j = 1; j <= i; j++) {
      row += j + " ";
    }
    console.log(row.trim());
  }
}
reverseNumberTriangle(5);

```

### **Explanation:**

#### **1. Outer Loop (i from N to 1):**

Each iteration represents a row, starting from N numbers down to 1.

#### **2. Inner Loop (j from 1 to i):**

Controls how many numbers are printed in the current row.

#### **3. Output:**

Numbers in each row start from 1 and decrease in length with each row.

## **6. Zigzag Pattern**

**Input:** N = 4

**Output:**

```
1 2 3 4
5 6 7 8
12 11 10 9
13 14 15 16
```

### **Code:**

```
function zigzagPattern(N) {
  let num = 1;
  for (let i = 1; i <= N; i++) {
    let row = [];
    for (let j = 1; j <= N; j++) {
      row.push(num++);
    }
    if (i % 2 === 0) row.reverse();
    console.log(row.join(" "));
  }
}
zigzagPattern(4);
```

### **Explanation:**

#### **1. Outer Loop (i from 1 to N):**

Each iteration represents a row.

#### **2. Row Construction:**

Add numbers to the row array sequentially.

### 3. Reversing Even Rows:

Reverse the array for even rows ( $i \% 2 == 0$ ) to create the zigzag effect.

### 4. Output:

Numbers are displayed row-wise with alternate rows reversed.

## 7. Spiral Matrix

**Input:**  $N = 4$

**Output:**

```
1 2 3 4
12 13 14 5
11 16 15 6
10 9 8 7
```

**Code:**

```
function spiralMatrix(N) {
  let matrix = Array.from({ length: N }, () => Array(N).fill(0));
  let num = 1,
    top = 0,
    bottom = N - 1,
    left = 0,
    right = N - 1;

  while (top <= bottom && left <= right) {
    for (let i = left; i <= right; i++) matrix[top][i] = num++;
    top++;
    for (let i = top; i <= bottom; i++) matrix[i][right] = num++;
    right--;
    for (let i = right; i >= left; i--) matrix[bottom][i] = num++;
    bottom--;
    for (let i = bottom; i >= top; i--) matrix[i][left] = num++;
    left++;
  }

  matrix.forEach((row) => console.log(row.join(" ")));
}
spiralMatrix(4);
```

**Explanation:**

### 1. Matrix Initialization:

Create a 2D array (matrix) of size  $N \times N$  filled with 0.

Variables top, bottom, left, right define the boundaries of the matrix.

## 2. Filling Spiral:

Start with the top row from left to right.

Move down the right column.

Fill the bottom row from right to left.

Finally, fill the left column from bottom to top.

## 3. Shrink Boundaries:

After filling each layer, move the boundaries inward (top++, bottom--, etc.).

## 4. Output:

The numbers are filled in a clockwise spiral order, and the final matrix is printed row by row.

## 8. Alphabet Pyramid

**Input:** N = 4

**Output:**

```
A  
B C  
D E F  
G H I J
```

**Code:**

```
function alphabetPyramid(N) {  
    let charCode = 65; // ASCII for 'A'  
    for (let i = 1; i <= N; i++) {  
        let row = "";  
        for (let j = 1; j <= i; j++) {  
            row += String.fromCharCode(charCode++) + " ";  
        }  
        console.log(row.trim());  
    }  
}  
alphabetPyramid(4);
```

**Explanation:**

## 1. Tracking Characters:

Use charCode starting from 65 (ASCII code for A) to represent alphabets.

Increment charCode after printing each letter.

## 2. Outer Loop (i from 1 to N):

Each iteration represents a row.

For i = 1, print 1 letter. For i = 2, print 2 letters, and so on.

## 3. Inner Loop (j from 1 to i):

Fill the current row with letters sequentially.

## 4. Output:

The pyramid displays letters row by row, starting with A and incrementing sequentially.

## 9. Full Square of Numbers

**Input:** N = 3

**Output:**

```
1 2 3  
4 5 6  
7 8 9
```

**Code:**

```
function numberSquare(N) {  
    let num = 1;  
    for (let i = 1; i <= N; i++) {  
        let row = "";  
        for (let j = 1; j <= N; j++) {  
            row += num++ + " ";  
        }  
        console.log(row.trim());  
    }  
}  
numberSquare(3);
```

**Explanation:**

## 1. Outer Loop (i from 1 to N):

Each iteration represents a row.

## 2. Inner Loop (j from 1 to N):

For each column, add the current num value to the row and increment num.

## 3. Tracking Numbers:

The num starts at 1 and increments sequentially as the matrix is filled.

#### 4. Output:

A full square matrix of size N x N is printed with consecutive numbers.

### 11. Hollow Square Pattern

**Input:** N = 5

**Output:**

```
* * * * *
*     *
*     *
*     *
* * * * *
```

**Code:**

```
function hollowSquare(N) {
  for (let i = 1; i <= N; i++) {
    if (i === 1 || i === N) {
      // Print full row of stars for the first and last rows
      console.log("* ".repeat(N).trim());
    } else {
      // Print a star, then spaces, then another star
      console.log("* " + " ".repeat(N - 2) + "*");
    }
  }
}
hollowSquare(5);
```

**Explanation:**

1. **First and Last Rows:** Use '`* .repeat(N)`' to print a full row of stars.
2. **Middle Rows:**

Start with \*.

Add (N-2) spaces in between.

End with another \*.

#### 3. Output:

First and last rows will be solid stars.

Middle rows will have stars only at the boundaries.

## 12. X Pattern

**Input:** N = 5

**Output:**

```
*      *
 *  *
 *
 *  *
*      *
```

**Code:**

```
function xPattern(N) {
  for (let i = 1; i <= N; i++) {
    let row = "";
    for (let j = 1; j <= N; j++) {
      if (j === i || j === N - i + 1) {
        row += "*";
      } else {
        row += " ";
      }
    }
    console.log(row);
  }
}
xPattern(5);
```

**Explanation:**

**1. Outer Loop (i from 1 to N):**

Each iteration represents a row.

**2. Inner Loop (j from 1 to N):**

Print \* for positions on the diagonals:

- j === i: Main diagonal (top-left to bottom-right).
- j === N - i + 1: Secondary diagonal (top-right to bottom-left).

Print spaces (' ') for other positions.

**3. Output:**

The stars form an "X" shape.

**Reel 16:**

## 1. Find Common Elements Between Two Arrays

**Question:** Given two arrays, find their common elements.

**Code:**

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];

let commonElements = arr1.filter(element => arr2.includes(element));
console.log(commonElements); // Output: [2, 1, 3]
```

**Explanation:**

1. **filter():**

Iterates through each element in arr1.

Checks if the element exists in arr2 using includes().

2. **includes():**

Returns true if the element exists in arr2, otherwise false.

3. **Result:**

The elements that return true are added to the commonElements array.

## 2. Find Common Elements Without Duplicates

**Question:** Find common elements, but ensure the result has no duplicates.

**Code:**

```
let arr1 = [2, 1, 4, 6, 3, 2];
let arr2 = [1, 7, 8, 3, 2, 2];

let commonElements = [...new Set(arr1.filter(element => arr2.includes(element)))];
console.log(commonElements); // Output: [2, 1, 3]
```

**Explanation:**

1. **Set:** Removes duplicates from the array.

2. **Process:**

Use filter() to find common elements.

Pass the result to new Set() to remove duplicates.

Convert back to an array using the spread operator [...].

### 3. Find Elements Unique to Each Array

*Question:* *Find elements that are unique to each array (not common).*

*Code:*

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];

let uniqueToArr1 = arr1.filter(element => !arr2.includes(element));
let uniqueToArr2 = arr2.filter(element => !arr1.includes(element));
let uniqueElements = [...uniqueToArr1, ...uniqueToArr2];

console.log(uniqueElements); // Output: [4, 6, 7, 8]
```

*Explanation:*

#### 1. Unique to arr1:

Use filter() to find elements in arr1 that do not exist in arr2.

#### 2. Unique to arr2:

Similarly, find elements in arr2 that do not exist in arr1.

#### 3. Combine Results:

Merge both arrays using the spread operator [...].

### 4. Find Union of Two Arrays

*Question:* *Combine two arrays and remove duplicates.*

*Code:*

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];

let union = [...new Set([...arr1, ...arr2])];
console.log(union); // Output: [2, 1, 4, 6, 3, 7, 8]
```

*Explanation:*

#### 1. Combine Arrays:

Use the spread operator to merge arr1 and arr2.

#### 2. Remove Duplicates:

Pass the merged array into new Set() to eliminate duplicates.

### 5. Find Intersection Using Sets

*Question:* Find common elements using Set for better performance.

*Code:*

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];

let set1 = new Set(arr1);
let intersection = arr2.filter(element => set1.has(element));

console.log(intersection); // Output: [1, 3, 2]
```

*Explanation:*

### 1. Create Set:

Convert arr1 into a Set to enable quick lookups.

### 2. Find Intersection:

Use filter() on arr2 and check if set1 contains the element.

## 6. Find Difference Between Two Arrays

*Question:* Find elements in arr1 that are not in arr2.

*Code:*

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];

let difference = arr1.filter(element => !arr2.includes(element));
console.log(difference); // Output: [4, 6]
```

*Explanation:*

### 1. Filter Elements:

Use filter() to find elements in arr1 that do not exist in arr2.

## 7. Find Symmetric Difference

*Question:* Find elements that are in either array but not in both.

*Code:*

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];

let diff1 = arr1.filter(element => !arr2.includes(element));
let diff2 = arr2.filter(element => !arr1.includes(element));
let symmetricDifference = [...diff1, ...diff2];

console.log(symmetricDifference); // Output: [4, 6, 7, 8]
```

### *Explanation:*

#### 1. Unique to arr1:

Find elements in arr1 not in arr2.

#### 2. Unique to arr2:

Find elements in arr2 not in arr1.

#### 3. Combine Results:

Merge the two results to get the symmetric difference.

## 8. Find Common Elements in Three Arrays

*Question: Find common elements in three arrays.*

*Code:*

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];
let arr3 = [3, 2, 9, 1];

let commonElements = arr1.filter(element => arr2.includes(element) &&
arr3.includes(element));
console.log(commonElements); // Output: [1, 3, 2]
```

### *Explanation:*

#### 1. Filter for All Arrays:

Use filter() to check if an element exists in both arr2 and arr3.

## 9. Check if Two Arrays Are Disjoint

*Question: Check if two arrays have no common elements.*

*Code:*

```
let arr1 = [2, 4, 6];
let arr2 = [1, 7, 8];

let isDisjoint = arr1.every(element => !arr2.includes(element));
console.log(isDisjoint); // Output: true
```

### *Explanation:*

#### 1. every():

Checks if every element in arr1 does not exist in arr2.

## 10. Count Common Elements

**Question:** Count the number of common elements between two arrays.

**Code:**

```
let arr1 = [2, 1, 4, 6, 3];
let arr2 = [1, 7, 8, 3, 2];

let commonCount = arr1.filter(element => arr2.includes(element)).length;
console.log(commonCount); // Output: 3
```

**Explanation:**

### 1. Find Common Elements:

Use filter() to find common elements.

### 2. Count:

Use .length to count the number of elements in the resulting array.

**Reel 17:**

### 1. Find the Largest Odd Number from a Numeric String

**Question:** Given a numeric string, return the smallest odd number that can be formed by using all or part of the digits

```
function largestOddNumber(num) {
    // Convert the string to an array of characters
    let numArray = num.split("");

    // Find the index of the last odd number
    let lastOddIndex = numArray.reverse().findIndex(digit => parseInt(digit) % 2 !== 0);

    // If no odd number is found, return an empty string
    if (lastOddIndex === -1) return "";

    // Reverse the index back to the original order
    lastOddIndex = num.length - lastOddIndex - 1;

    // Slice the string from the start to the last odd number
    return num.slice(0, lastOddIndex + 1);
}

// Test Cases
console.log(largestOddNumber("5688248")); // Output: "5"
console.log(largestOddNumber("61632826")); // Output: "6163"
```

```
console.log(largestOddNumber("616338261")); // Output: "616338261"
```

### Explanation:

#### 1. Convert to Array:

split("") : Converts the numeric string into an array of individual characters.

#### 2. Reverse the Array:

reverse() : Reverses the array so that we start checking for odd digits from the rightmost side (end of the string).

#### 3. Find the Last Odd Digit:

findIndex() : Locates the first odd digit in the reversed array. This represents the last odd digit in the original string.

#### 4. Adjust the Index:

Since the array was reversed, convert the index back to the original order with:

```
lastOddIndex = num.length - lastOddIndex - 1;
```

Extract the Substring:

slice(0, lastOddIndex + 1) : Extracts all characters from the start of the string up to the last odd digit.

#### 5. Handle Edge Cases:

If findIndex() returns -1 (no odd digits found), return an empty string.

### Output:

For the test cases:

- num = "5688248" → Output: "5"
- num = "61632826" → Output: "6163"
- num = "616338261" → Output: "616338261"

## 2. Find the Smallest Odd Number from a Numeric String

**Question:** Given a numeric string, return the smallest odd number that can be formed by using all or part of the digits.

```
function smallestOddNumber(num) {  
    // Convert the string into an array of digits  
    let numArray = num.split("").map(Number);
```

```

// Filter only the odd digits and sort them
let oddDigits = numArray.filter(digit => digit % 2 !== 0).sort((a, b) => a - b);

// If no odd digits are found, return an empty string
return oddDigits.length > 0 ? String(oddDigits[0]) : "";
}

// Test Cases
console.log(smallestOddNumber("5688248")); // Output: "5"
console.log(smallestOddNumber("61632826")); // Output: "3"
console.log(smallestOddNumber("246802")); // Output: "" (no odd digits)

```

### ***Explanation:***

#### **1. Convert String to Array:**

Use .split("") to break the numeric string into an array of characters.

Convert each character back into a number using .map(Number).

#### **2. Filter Odd Digits:**

Use .filter() to extract only the digits that are odd (digit % 2 !== 0).

#### **3. Sort Odd Digits:**

Use .sort((a, b) => a - b) to sort the odd digits in ascending order.

#### **4. Check for Odd Digits:**

If the filtered array is empty (oddDigits.length === 0), return an empty string.

#### **5. Return the Smallest Odd Digit:**

Return the first element of the sorted array (oddDigits[0]).

### **3. Largest Even Number from Left to Right**

**Question:** Given a numeric string, find the largest *even number* that can be formed by using digits sequentially from left to right.

```

function largestEvenNumber(num) {
  for (let i = num.length - 1; i >= 0; i--) {
    if (parseInt(num[i]) % 2 === 0) {
      return num.slice(0, i + 1);
    }
  }
}

```

```

        }
        return ""; // No even number found
    }

// Test Cases
console.log(largestEvenNumber("5688248")); // Output: "5688248"
console.log(largestEvenNumber("61632826")); // Output: "61632826"
console.log(largestEvenNumber("13579")); // Output: "" (no even digits)

```

**Explanation:**

**1. Traverse from Right to Left:**

Use a for loop starting from the end (`num.length - 1`) and move backward (`i--`).

**2. Check for Even Digits:**

Use `parseInt(num[i]) % 2 === 0` to check if the current digit is even.

**3. Return Largest Even Substring:**

When an even digit is found, use `.slice(0, i + 1)` to include all characters from the start to the current position.

This ensures the largest even number is returned.

**4. Handle No Even Digits:**

If no even digits are found, the function will return an empty string.

**4. Count the Number of Odd Digits**

**Question:** **Count how many odd digits are present in a numeric string.**

```

function countOddDigits(num) {
    // Convert the string to an array and filter odd digits
    return num.split("").filter(digit => parseInt(digit) % 2 !== 0).length;
}

// Test Cases
console.log(countOddDigits("5688248")); // Output: 2 (5 and 3 are odd)
console.log(countOddDigits("61632826")); // Output: 2 (3 and 1 are odd)
console.log(countOddDigits("246802")); // Output: 0 (no odd digits)

```

**Explanation:**

1. **Split the String:**
    - o Use `.split("")` to convert the numeric string into an array of individual characters.
  2. **Filter Odd Digits:**
    - o Use `.filter(digit => parseInt(digit) % 2 !== 0)` to keep only odd digits.
  3. **Count the Odd Digits:**
    - o Use `.length` to count how many elements are in the filtered array.
5. Sum of All Odd Digits

**Question:** **Find the sum of all odd digits in a numeric string.**

```
function sumOfOddDigits(num) {  
    // Convert the string to an array, filter odd digits, and sum them  
    return num.split("")  
        .map(Number)  
        .filter(digit => digit % 2 !== 0)  
        .reduce((sum, digit) => sum + digit, 0);  
}  
  
// Test Cases  
console.log(sumOfOddDigits("5688248")); // Output: 8 (5 + 3)  
console.log(sumOfOddDigits("61632826")); // Output: 4 (3 + 1)  
console.log(sumOfOddDigits("246802")); // Output: 0 (no odd digits)
```

**Explanation:**

1. **Convert String to Array:**

Use `.split("")` to convert the string into an array of characters.

Convert each character to a number using `.map(Number)`.

2. **Filter Odd Digits:**

Use `.filter(digit => digit % 2 !== 0)` to retain only the odd digits.

3. **Sum Odd Digits:**

Use `.reduce((sum, digit) => sum + digit, 0):`

- Start with an initial sum of 0.
- Add each odd digit to the sum.

## Reel 18:

### 1. Convert Roman Numerals to Integer

**Question: Write a program to convert a Roman numeral string into its corresponding integer value. The program should follow the rules of Roman numeral representation and handle cases where subtraction is required (e.g., IV = 4, IX = 9).**

**Code:**

```
function romanToInt(s) {  
    // Roman numeral to integer mapping  
    const romanMap = {  
        'I': 1,  
        'V': 5,  
        'X': 10,  
        'L': 50,  
        'C': 100,  
        'D': 500,  
        'M': 1000  
    };  
  
    // Convert the string into an array and map values  
    const values = s.split("").map(char => romanMap[char]);  
  
    // Reduce the array into a single integer  
    return values.reduce((sum, currentValue, index) => {  
        // If the current value is less than the next value, subtract it  
        if (index < values.length - 1 && currentValue < values[index + 1]) {  
            return sum - currentValue;  
        }  
        // Otherwise, add it  
        return sum + currentValue;  
    }, 0);  
}  
  
// Test Cases  
console.log(romanToInt("LVIII")); // Output: 58  
console.log(romanToInt("IX")); // Output: 9  
console.log(romanToInt("MCMXCIV")); // Output: 1994
```

### Explanation

#### Step 1: Map Roman Numerals to Integer Values

- Use `s.split("")` to convert the string into an array of characters.
- Use `map()` to transform each character into its corresponding integer value using the `romanMap` object.
- Example: For "LVIII", values will be [50, 5, 1, 1, 1].

## Step 2: Apply the Subtraction Rule with reduce()

- Use reduce() to iterate over the array of values and calculate the final integer:
  - If the current value is smaller than the next value, subtract it from the sum (e.g., I before X becomes -1).
  - Otherwise, add the current value to the sum.
- Example for "LVIII":
  - $50 + 5 + 1 + 1 + 1 = 58$ .

## Step 3: Handle Edge Cases

- The loop stops when all values are processed.
- If there's no next value (index === values.length - 1), always add the current value.

## 2. Integer to Roman

*Question:* Convert an integer to its Roman numeral representation.

*Code:*

```
function intToRoman(num) {  
    const romanMap = [  
        { value: 1000, symbol: "M" },  
        { value: 900, symbol: "CM" },  
        { value: 500, symbol: "D" },  
        { value: 400, symbol: "CD" },  
        { value: 100, symbol: "C" },  
        { value: 90, symbol: "XC" },  
        { value: 50, symbol: "L" },  
        { value: 40, symbol: "XL" },  
        { value: 10, symbol: "X" },  
        { value: 9, symbol: "IX" },  
        { value: 5, symbol: "V" },  
        { value: 4, symbol: "IV" },  
        { value: 1, symbol: "I" }  
    ];  
  
    let result = "";  
  
    for (const { value, symbol } of romanMap) {  
        while (num >= value) {  
            result += symbol;  
            num -= value;  
        }  
    }  
  
    return result;  
}  
  
// Test Cases  
console.log(intToRoman(58)); // Output: "LVIII"  
console.log(intToRoman(1994)); // Output: "MCMXCIV"
```

## **Explanation:**

### **1. Roman Mapping:**

Use a sorted mapping of Roman numerals and their values, starting from the largest ( $M = 1000$ ) to the smallest ( $I = 1$ ).

### **2. Loop Through Mapping:**

For each Roman numeral, check if the current number (num) is greater than or equal to the value:

- If num  $\geq$  value, add the symbol to result and subtract value from num.
- Repeat until num is less than value.

### **3. Stop When Done:**

Continue this process until the entire number is converted.

## **3. Valid Parentheses**

*Question: Determine if a string containing (), {}, and [] is valid.*

**Code:**

```
function isValidParentheses(s) {  
    const stack = [];  
    const pairs = { ')': '(', '}': '{', ']': '[' };  
  
    for (const char of s) {  
        if (char in pairs) {  
            if (stack.pop() !== pairs[char]) {  
                return false;  
            }  
        } else {  
            stack.push(char);  
        }  
    }  
  
    return stack.length === 0;  
}  
  
// Test Cases  
console.log(isValidParentheses("()")); // Output: true  
console.log(isValidParentheses("()[]{}")); // Output: true  
console.log(isValidParentheses("[]")); // Output: false
```

## **Explanation:**

### **1. Pairs Mapping:**

Use an object pairs to map closing brackets () → (, } → {, etc.).

## 2. Use a Stack:

Push opening brackets ((, {, [) onto the stack.

When encountering a closing bracket (, }, ]), check if it matches the **top of the stack**:

- Use stack.pop() to remove the last opened bracket.
- Compare it with the matching bracket from pairs.

## 3. Return Validity:

If the stack is empty at the end, the string is valid.

If the stack is not empty, it means there are unmatched brackets.

## 4. Longest Substring Without Repeating Characters

*Question: Find the length of the longest substring without repeating characters.*

*Code:*

```
function lengthOfLongestSubstring(s) {
    const seen = new Set();
    let left = 0, maxLength = 0;

    for (let right = 0; right < s.length; right++) {
        while (seen.has(s[right])) {
            seen.delete(s[left]);
            left++;
        }
        seen.add(s[right]);
        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}

// Test Cases
console.log(lengthOfLongestSubstring("abcabcbb")); // Output: 3 ("abc")
```

**Explanation:**

### 1. Sliding Window:

Use two pointers (left and right) to represent a sliding window of characters.

### 2. Set for Uniqueness:

Use a Set to store unique characters in the current window.

### 3. Expand and Shrink Window:

Expand the window by moving right.

Shrink the window by moving left until the substring has no duplicates.

#### 4. Track Maximum Length:

Update maxLength for each valid substring.

### 5. String Compression

*Question: Compress an array of characters in place by replacing consecutive duplicates with the character and count.*

*Code:*

```
function compress(chars) {  
    let write = 0, read = 0;  
  
    while (read < chars.length) {  
        const char = chars[read];  
        let count = 0;  
  
        while (read < chars.length && chars[read] === char) {  
            read++;  
            count++;  
        }  
  
        chars[write++] = char;  
  
        if (count > 1) {  
            for (const digit of String(count)) {  
                chars[write++] = digit;  
            }  
        }  
    }  
  
    return write;  
}  
  
// Test Cases  
const chars1 = ["a", "a", "b", "b", "c", "c", "c"];  
console.log(compress(chars1)); // Output: 6  
console.log(chars1.slice(0, 6)); // ["a", "2", "b", "2", "c", "3"]
```

#### Explanation:

##### 1. Two Pointers:

read: Reads through the original array.

write: Writes the compressed version in place.

## 2. Count Duplicates:

Count consecutive duplicates using a nested loop.

## 3. Write Characters and Counts:

Write the character.

Write the count if greater than 1.

## Reel 19:

### Find Top K Frequent Elements

**Question: Given an integer array nums and an integer k, return the k most frequent elements.**

Input: nums = [1, 1, 1, 2, 2, 3], k = 2  
Output: [1, 2]

Input: nums = [4, 4, 4, 6, 6, 6, 6, 7, 8], k = 1  
Output: [6]

### Code:

```
function topKFrequent(nums, k) {  
    // Step 1: Count frequencies using reduce  
    const frequencyMap = nums.reduce((acc, num) => {  
        acc[num] = (acc[num] || 0) + 1;  
        return acc;  
    }, {});  
  
    // Step 2: Sort the elements by frequency  
    const sortedFrequencies = Object.entries(frequencyMap).sort(  
        (a, b) => b[1] - a[1]  
    ); // Sort by frequency in descending order  
  
    // Step 3: Extract the top k elements  
    return sortedFrequencies.slice(0, k).map(([num]) => parseInt(num));  
}
```

Explanation:

1. **Reduce (Step 1):** Create a frequency map from the input array. For example:

Input: [1, 1, 1, 2, 2, 3]

Output: { 1: 3, 2: 2, 3: 1 }

2. **Sort (Step 2):** Convert the frequency map into an array of key-value pairs and sort it by frequency:

Input: Object.entries({ 1: 3, 2: 2, 3: 1 })

Output (sorted): [ [ '1', 3 ], [ '2', 2 ], [ '3', 1 ] ]

3. **Slice and Map (Step 3):** Take the top k elements and map them to their numeric keys:

Input: [[ '1', 3 ], [ '2', 2 ]]

Output: [1, 2]

Rotate an Array

**Question: Write a function to rotate an array to the right by k steps, where k is non-negative.**

Input: (nums = [1, 2, 3, 4, 5, 6, 7]), (k = 3);  
Output: [5, 6, 7, 1, 2, 3, 4];

Code:

```
function rotateArray(nums, k) {
  k = k % nums.length; // Handle cases where k > nums.length
  return [...nums.slice(-k), ...nums.slice(0, -k)]; // Use slice and spread
}

// Example run:
const nums = [1, 2, 3, 4, 5, 6, 7];
console.log(rotateArray(nums, 3)); // Output: [5, 6, 7, 1, 2, 3, 4]
```

Explanation:

### 1. Understanding Rotation:

When rotating to the right by k steps, the last k elements move to the front, and the remaining elements shift to the right.

Example: nums = [1, 2, 3, 4, 5, 6, 7], k = 3

- Last k = 3 elements: [5, 6, 7]
- Remaining elements: [1, 2, 3, 4]
- Result: [5, 6, 7, 1, 2, 3, 4].

### 2. Modulo Operation (k % nums.length):

If k > nums.length, rotating the array by k steps is the same as rotating it by k % nums.length.

Example: For `nums = [1, 2, 3]`, rotating  $k = 4$  steps is the same as rotating  $k = 1$  step.

### 3. Using `slice()`:

`nums.slice(-k)`: Extracts the last  $k$  elements.

`nums.slice(0, -k)`: Extracts the remaining elements from the start of the array up to the last  $k$  elements.

### 4. Combining with the Spread Operator:

Combine the two slices using the spread operator ... to create the rotated array.

Find the First Missing Positive Integer

**Question: Write a function to find the smallest positive integer that is missing from an array.**

**Input:** `nums = [3, 4, -1, 1];`  
**Output:** 2;

Code:

```
function firstMissingPositive(nums) {  
    // Step 1: Filter positive numbers and sort them  
    nums = nums.filter((num) => num > 0).sort((a, b) => a - b);  
  
    // Step 2: Find the first missing positive integer  
    let missing = 1;  
    nums.forEach((num) => {  
        if (num === missing) {  
            missing++;  
        }  
    });  
  
    return missing;  
}  
  
// Example run:  
const nums = [3, 4, -1, 1];  
console.log(firstMissingPositive(nums)); // Output: 2
```

Explanation:

#### 1. Filter Positive Numbers:

Use filter( $\text{num} \Rightarrow \text{num} > 0$ ) to remove all non-positive numbers.

Example: For  $\text{nums} = [3, 4, -1, 1]$ , the filtered array becomes  $[3, 4, 1]$ .

## 2. Sort the Array:

$\text{sort}((a, b) \Rightarrow a - b)$ : Sorts the filtered array in ascending order.

Example:  $[3, 4, 1]$  becomes  $[1, 3, 4]$ .

## 3. Find the Missing Positive:

Use forEach to iterate over the sorted array.

Start with  $\text{missing} = 1$  (the smallest positive integer).

If  $\text{num} === \text{missing}$ , increment  $\text{missing}$  by 1.

Example:

- Array:  $[1, 3, 4]$
- Iteration 1:  $\text{num} = 1$ , so  $\text{missing} = 2$ .
- Iteration 2:  $\text{num} = 3$ ,  $\text{missing} = 2$  is not updated.
- Iteration 3:  $\text{num} = 4$ ,  $\text{missing} = 2$  is still missing.

## 4. Return the Result:

The first missing positive is 2.

## Reel 20:

### Question: Group Anagrams

**Question:** You are given an array of strings. Write a program to group anagrams together.

#### **Input:**

```
strs = ["eat", "tea", "tan", "ate", "nat", "bat"];
```

#### **Output:**

```
[["eat", "tea", "ate"], ["tan", "nat"], ["bat"]];
```

#### **Explanation:**

##### 1. Sorting the Strings:

For each word, split it into characters, sort them alphabetically, and join them back into a string.

Example:

- "eat" → "aet"
- "tea" → "aet"
- "tan" → "ant"

## 2. Using a Map:

Use the sorted string as a key in the map.

Group words with the same sorted string into an array.

## 3. Extracting the Groups:

Once the map is built, use Object.values(map) to extract the grouped anagrams.

### **Count Total Anagram Pairs in an Array**

**Question:** Given an array of strings, find the total number of anagram pairs.

Input: ["listen", "silent", "enlist", "hello", "world"];  
Output: 3;

**Code:**

```
function countAnagramPairs(words) {  
    const sortedWords = words.map((word) => word.split("").sort().join(""));  
  
    const frequencyMap = sortedWords.reduce((acc, word) => {  
        acc[word] = (acc[word] || 0) + 1;  
        return acc;  
    }, {});  
  
    return Object.values(frequencyMap).reduce((pairs, count) => {  
        return pairs + (count * (count - 1)) / 2;  
    }, 0);  
}  
  
console.log(  
    countAnagramPairs(["listen", "silent", "enlist", "hello", "world"])  
);  
// Output: 3
```

### **Explanation:**

1. **Sorting Words:** Each word is sorted alphabetically. This ensures all anagrams are represented the same way.
  - "listen" → "eilnst"
  - "silent" → "eilnst"
  - "enlist" → "eilnst"

- "hello" → "ehllo"
  - "world" → "dlorw"
2. **Frequency Map:** Count the number of times each sorted word appears:
    - { "eilnst": 3, "ehllo": 1, "dlorw": 1 }
  3. **Calculate Anagram Pairs:** If a word appears n times, the number of pairs is calculated as:
    - $n * (n - 1) / 2$
    - For "eilnst", there are 3 words, so pairs =  $3 * (3 - 1) / 2 = 3$ .
  4. **Output:** The total pairs across all groups are summed, giving the result: 3.

### Find the Largest Group of Anagrams

**Question:** Given an array of strings, find the largest group of anagrams and return it.

```
Input: ["eat", "tea", "tan", "ate", "nat", "bat"];
Output: ["eat", "tea", "ate"];
```

**Code:**

```
function largestAnagramGroup(words) {
  const map = {};

  words.forEach((word) => {
    const sorted = word.split("").sort().join("");
    if (!map[sorted]) map[sorted] = [];
    map[sorted].push(word);
  });

  const largestGroup = Object.values(map).reduce((largest, group) =>
    group.length > largest.length ? group : largest
  );

  return largestGroup;
}

console.log(largestAnagramGroup(["eat", "tea", "tan", "ate", "nat", "bat"]));
// Output: ["eat", "tea", "ate"]
```

**Explanation:**

1. **Grouping Anagrams:** Sort each word to use as a key, and group the original words in a map:
  - { "aet": ["eat", "tea", "ate"], "ant": ["tan", "nat"], "abt": ["bat"] }
2. **Find the Largest Group:** Compare the size of each group and keep the largest one:
  - Largest group: ["eat", "tea", "ate"]
3. **Output:** Return the largest group of anagrams.

### Check if Two Strings Are K-Anagrams

**Question:** Two strings are **k-anagrams** if you can convert one string into the other by changing at most k characters. Write a function to check if two strings are k-anagrams.

Input: (s1 = "anagram"), (s2 = "mangaar"), (k = 1);

Output: true;

Input: (s1 = "anagram"), (s2 = "managra"), (k = 0);

Output: false;

**Code:**

```
function areKAnagrams(s1, s2, k) {  
    if (s1.length !== s2.length) return false;  
  
    const count1 = {};  
    const count2 = {};  
  
    for (let char of s1) count1[char] = (count1[char] || 0) + 1;  
    for (let char of s2) count2[char] = (count2[char] || 0) + 1;  
  
    let changes = 0;  
  
    for (let char in count1) {  
        if (count1[char] > count2[char] || !count2[char]) {  
            changes += count1[char] - (count2[char] || 0);  
        }  
    }  
  
    return changes <= k;  
}  
  
console.log(areKAnagrams("anagram", "mangaar", 1)); // Output: true  
console.log(areKAnagrams("anagram", "managra", 0)); // Output: false
```

**Explanation:**

1. **Count Character Frequencies:** Create frequency maps for both strings:
  - o For "anagram": { a: 3, n: 1, g: 1, r: 1, m: 1 }
  - o For "mangaar": { m: 2, a: 3, n: 1, g: 1, r: 1 }
2. **Calculate Differences:** Compare the maps:
  - o Count how many characters need to change to make the maps match.
3. **Check k:** If the total changes needed are  $\leq k$ , return true; otherwise, return false.

**Reel 21:**

**Question 1: Find First Non-Repeating Character**

**Problem:**

You are given a string. Write a program to find the **first non-repeating character** in the string. If all characters repeat, return null.

Input: "javascript";

Output: "j";

Input: "aabbcc";

Output: null;

Input: "swiss";

Output: "w";

### Solution Code:

```
function firstNonRepeatingChar(str) {  
    const charCount = {};  
  
    // Count occurrences of each character  
    for (let char of str) {  
        charCount[char] = (charCount[char] || 0) + 1;  
    }  
  
    // Find the first character with a count of 1  
    for (let char of str) {  
        if (charCount[char] === 1) {  
            return char;  
        }  
    }  
  
    return null; // If no non-repeating character is found  
}  
  
console.log(firstNonRepeatingChar("javascript")); // Output: "j"  
console.log(firstNonRepeatingChar("aabbcc")); // Output: null  
console.log(firstNonRepeatingChar("swiss")); // Output: "w"
```

### Explanation:

#### Step 1: Count Characters:

Create an object to count how many times each character appears in the string.

Example:

- For "javascript", the count object will look like:  
{ j: 1, a: 2, v: 1, s: 1, c: 1, r: 1, i: 1, p: 1, t: 1 }

#### Step 2: Find the First Non-Repeating Character:

Loop through the string again and check the count of each character.

- The first character with a count of 1 is our answer.

#### If No Characters Are Non-Repeating:

If no character has a count of 1, return null.

## Question 2: Reverse a String

*Problem:*

Write a function to reverse a string.

*Input and Output:*

Input: "javascript";  
Output: "tpircsavaj";

Code:

```
function reverseString(str) {  
  return str.split("").reverse().join("");  
  
}  
  
console.log(reverseString("javascript")); // Output: "tpircsavaj"
```

**Explanation:**

1. **Split:** The `split("")` method splits the string into an array of characters.
  - o "javascript" → ["j", "a", "v", "a", "s", "c", "r", "i", "p", "t"]
2. **Reverse:** The `reverse()` method reverses the array.
  - o [ "t", "p", "i", "r", "c", "s", "a", "v", "a", "j" ]
3. **Join:** The `join("")` method combines the reversed array into a single string.

## Question 3: Check if a String is a Palindrome

*Problem:*

Write a function to check if a string reads the same backward as forward.

*Input and Output:*

Input: "madam";  
Output: true;  
Input: "hello";  
Output: false;

Code:

```
function isPalindrome(str) {  
  const reversed = str.split("").reverse().join("");  
  return str === reversed;  
}
```

```
console.log(isPalindrome("madam")); // Output: true  
console.log(isPalindrome("hello")); // Output: false
```

### Explanation:

1. Reverse the string and compare it to the original string.
2. If they are equal, the string is a palindrome.

Question 4: Count Vowels in a String

*Problem:*

Write a function to count the number of vowels in a string.

*Input and Output:*

```
Input: "javascript";  
Output: 3;
```

### Solution Code:

```
function countVowels(str) {  
  const vowels = "aeiouAEIOU";  
  return str.split("").filter((char) => vowels.includes(char)).length;  
}  
  
console.log(countVowels("javascript")); // Output: 3
```

### Explanation:

1. Split the string into characters.
2. Use filter() to count characters that match vowels.
3. Return the count.

Question 5: Find the First Non-Repeating Character

*Problem:*

Find the first character in a string that does not repeat.

*Input and Output:*

```
Input: "swiss";  
Output: "w";
```

```
Input: "aabbb";  
Output: null;
```

**Code:**

```
function firstNonRepeatingChar(str) {  
    const charCount = {};  
    for (let char of str) {  
        charCount[char] = (charCount[char] || 0) + 1;  
    }  
  
    for (let char of str) {  
        if (charCount[char] === 1) {  
            return char;  
        }  
    }  
  
    return null;  
}  
  
console.log(firstNonRepeatingChar("swiss")); // Output: "w"  
console.log(firstNonRepeatingChar("aabb")); // Output: null
```

**Explanation:**

1. Use an object to count the frequency of each character.
2. Iterate over the string again to find the first character with a count of 1.

Question 6: Check if Two Strings Are Anagrams

**Problem:**

Check if two strings are anagrams of each other.

**Input and Output:**

Input: "listen", "silent";  
Output: true;

Input: "hello", "world";  
Output: false;

**Solution Code:**

```
function isAnagram(s1, s2) {  
    if (s1.length !== s2.length) return false;  
    return s1.split("").sort().join("") === s2.split("").sort().join("");  
}  
  
console.log(isAnagram("listen", "silent")); // Output: true  
console.log(isAnagram("hello", "world")); // Output: false
```

**Explanation:**

1. Sort the characters in both strings.

2. Compare the sorted versions. If they match, the strings are anagrams.

### Question 7: Remove Duplicates from a String

*Problem:*

Remove all duplicate characters from a string.

*Input and Output:*

```
Input: "javascript";
Output: "javscript";
```

### Solution Code:

```
function removeDuplicates(str) {
  return [...new Set(str)].join("");
}

console.log(removeDuplicates("javascript")); // Output: "javscript"
```

### Explanation:

1. Use new Set() to remove duplicate characters.
2. Convert the Set back to a string using join("")�.

### Question 8: Check if a String is a Subsequence of Another

*Problem:*

Check if one string is a subsequence of another string.

*Input and Output:*

```
Input: (s1 = "abc"), (s2 = "aabbcc");
Output: true;
```

```
Input: (s1 = "aec"), (s2 = "abcde");
Output: false;
```

### Solution Code:

```
function isSubsequence(s1, s2) {
  let i = 0;
  for (let char of s2) {
    if (char === s1[i]) i++;
    if (i === s1.length) return true;
  }
  return false;
}
```

```
console.log(isSubsequence("abc", "aabbcc")); // Output: true
console.log(isSubsequence("aec", "abcde")); // Output: false
```

### Explanation:

1. Iterate through s2 while checking if characters match the order in s1.
2. Return true if all characters in s1 are found in order.

## Question 9: Capitalize the First Letter of Each Word

*Problem: Capitalize the first letter of each word in a string.*

*Input and Output:*

```
Input: "i love javascript";
Output: "I Love JavaScript";
```

**Code:**

```
function capitalizeWords(str) {
  return str
    .split(" ")
    .map((word) => word[0].toUpperCase() + word.slice(1))
    .join(" ");
}

console.log(capitalizeWords("i love javascript")); // Output: "I Love JavaScript"
```

### Explanation:

1. Split the string into words.
2. Use map() to capitalize the first letter of each word.
3. Join the words back into a string.

## Reel 22:

### Question 1:

*Write a JavaScript function to find the second largest number in an array.*

**Code:**

```
function findSecondLargest(arr) {
  let largest = -Infinity;
  let secondLargest = -Infinity;

  for (let num of arr) {
    if (num > largest) {
      secondLargest = largest; // Update second largest
      largest = num; // Update largest
```

```

} else if (num > secondLargest && num < largest) {
    secondLargest = num; // Update only second largest
}
}

return secondLargest === -Infinity ? null : secondLargest;
}

console.log(findSecondLargest([10, 20, 4, 45, 99])); // Output: 45
console.log(findSecondLargest([3, 3, 3, 3])); // Output: null

```

### **Explanation:**

- The function initializes largest and secondLargest as -Infinity to handle negative numbers.
- It iterates through the array, updating largest and secondLargest based on conditions:
  - If the current number is greater than largest, update both largest and secondLargest.
  - If it's between largest and secondLargest, update only secondLargest.
- At the end, if secondLargest is still -Infinity, it returns null as there's no second largest number.

### Question 2:

**What edge cases should be considered while finding the second largest number in an array?**

### **Answer:**

#### **Empty Array:**

If the input array is empty, return null.

```
console.log(findSecondLargest([])); // Output: null
```

#### **Single Element Array:**

If the array has only one element, there is no second largest number.

```
console.log(findSecondLargest([5])); // Output: null
```

#### **Array with All Equal Numbers:**

If all numbers are the same, there's no second largest number.

```
console.log(findSecondLargest([3, 3, 3, 3])); // Output: null
```

#### **Negative Numbers:**

Ensure the function works with negative numbers.

```
console.log(findSecondLargest([-10, -20, -30, -5])); // Output: -10
```

### Mixed Numbers:

Works correctly for mixed positive and negative numbers.

```
console.log(findSecondLargest([10, -20, 30, 15])); // Output: 15
```

### Question 3:

What is the time complexity of the `findSecondLargest` function? Can it be optimized?

### Answer:

- **Time Complexity:** The function has a single loop, so the time complexity is **O(n)**, where n is the size of the input array.
- **Space Complexity:** It uses two variables (`largest` and `secondLargest`), so the space complexity is **O(1)**.

### Question 4:

How can you handle arrays where duplicates exist but still find the second largest?

### Answer:

The current implementation already handles duplicates by ignoring them. For example:

```
console.log(findSecondLargest([10, 20, 20, 45, 99])); // Output: 45
```

If the array has repeated elements and you want to explicitly filter them out before processing, you can use the `Set` object to remove duplicates:

```
function findSecondLargestUnique(arr) {  
  const uniqueArr = [...new Set(arr)];  
  return findSecondLargest(uniqueArr);  
}  
  
console.log(findSecondLargestUnique([10, 20, 20, 45, 99])); // Output: 45
```

### Question 5:

What will happen if the input array contains non-numeric values?

### Answer:

If the array contains non-numeric values, they can cause unexpected results. For example:

```
console.log(findSecondLargest([10, "20", 4, 45, "abc"])); // Output: NaN
```

To handle this, filter the array to include only numbers:

```
function findSecondLargestNumbersOnly(arr) {  
  const filteredArr = arr.filter((num) => typeof num === "number");  
}
```

```
    return findSecondLargest(filteredArr);
}

console.log(findSecondLargestNumbersOnly([10, "20", 4, 45, "abc"])); // Output: 45
```

### Explanation:

- filter() ensures only numeric values are passed to the main logic.
- This prevents issues caused by strings or other data types.

## Reel 23:

### Question 1: Find the Missing Number in an Array

#### Problem:

You are given an array of integers from 1 to n, but one number is missing. Write a program to find the missing number.

#### Input and Output:

Input: [1, 2, 4, 5]

Output: 3

Input: [3, 7, 1, 2, 8, 4, 5]

Output: 6

Input: [1, 2, 3, 4, 5]

Output: null (No missing number)

#### Solution Code:

```
function findMissingNumber(arr, n) {
  const expectedSum = (n * (n + 1)) / 2; // Sum of first n natural numbers
  const actualSum = arr.reduce((acc, num) => acc + num, 0); // Sum of array elements
  return expectedSum === actualSum ? null : expectedSum - actualSum;
}

console.log(findMissingNumber([1, 2, 4, 5], 5)); // Output: 3
console.log(findMissingNumber([3, 7, 1, 2, 8, 4, 5], 8)); // Output: 6
console.log(findMissingNumber([1, 2, 3, 4, 5], 5)); // Output: null
```

#### Calculate the Expected Sum:

The sum of the first n natural numbers is  $(n * (n + 1)) / 2$ .

### Calculate the Actual Sum:

Use reduce() to calculate the sum of the array elements.

### Find the Missing Number:

Subtract the actual sum from the expected sum.

- If the sums are equal, return null.
- Otherwise, return the difference (missing number).

## Question 2: Find Two Missing Numbers in an Array

### *Problem:*

You are given an array of integers from 1 to n, but two numbers are missing. Write a program to find the two missing numbers.

### *Input and Output:*

Input: [1, 2, 4, 6], (n = 6);

Output: [3, 5];

### Solution Code:

```
function findTwoMissingNumbers(arr, n) {  
    const totalSum = (n * (n + 1)) / 2; // Sum of first n natural numbers  
    const totalSquareSum = (n * (n + 1) * (2 * n + 1)) / 6; // Sum of squares of first n natural  
    numbers  
  
    const actualSum = arr.reduce((acc, num) => acc + num, 0);  
    const actualSquareSum = arr.reduce((acc, num) => acc + num * num, 0);  
  
    const sumOfMissing = totalSum - actualSum; // x + y  
    const squareSumOfMissing = totalSquareSum - actualSquareSum; // x^2 + y^2  
  
    const productOfMissing = (sumOfMissing ** 2 - squareSumOfMissing) / 2; // x * y  
  
    const x =  
        (sumOfMissing + Math.sqrt(sumOfMissing ** 2 - 4 * productOfMissing)) / 2;  
    const y = sumOfMissing - x;  
  
    return [x, y];  
}  
  
console.log(findTwoMissingNumbers([1, 2, 4, 6], 6)); // Output: [3, 5]
```

### **Explanation:**

#### **1. Calculate the Total Sum and Square Sum:**

Use the formulas  $(n * (n + 1)) / 2$  for the sum and  $(n * (n + 1) * (2 * n + 1)) / 6$  for the square sum of the first  $n$  numbers.

#### **2. Find the Differences:**

Subtract the actual sums (from the array) from the total sums to get:

- $\text{sumOfMissing} = x + y$
- $\text{squareSumOfMissing} = x^2 + y^2$

#### **3. Solve for the Two Missing Numbers:**

Using algebra:

- $x * y = (\text{sumOfMissing}^2 - \text{squareSumOfMissing}) / 2$
- Solve the quadratic equation to find  $x$  and  $y$ .

#### **4. Output the Result:**

Return  $[x, y]$ .

## **Question 3: Find the Missing Number in an Unsorted Array**

### **Problem:**

You are given an unsorted array of integers from 1 to  $n$ , but one number is missing. Find the missing number without sorting the array.

### ***Input and Output:***

Input: [4, 3, 1, 5];

Output: 2;

### **Code:**

```
function findMissingUnsorted(arr, n) {  
    const totalSum = (n * (n + 1)) / 2; // Sum of first n natural numbers  
    const actualSum = arr.reduce((acc, num) => acc + num, 0);  
    return totalSum - actualSum;  
}  
  
console.log(findMissingUnsorted([4, 3, 1, 5], 5)); // Output: 2
```

### **Explanation:**

### **1. Calculate the Total Sum:**

Use  $(n * (n + 1)) / 2$  to find the sum of numbers from 1 to n.

### **2. Calculate the Actual Sum:**

Use reduce() to calculate the sum of all numbers in the array.

### **3. Find the Missing Number:**

Subtract the actual sum from the total sum to find the missing number.

### **4. Output:**

For the array [4, 3, 1, 5] and n = 5, the missing number is 2.

## **Question 4: Find the Missing Number in a Range with Duplicates**

### ***Problem:***

You are given an array of integers, some of which are repeated, and one number is missing. Write a program to find the missing number.

### **Input and Output:**

Input: [1, 2, 2, 4, 5];

Output: 3;

### **Solution Code:**

```
function findMissingWithDuplicates(arr, n) {  
  const uniqueSum = [...new Set(arr)].reduce((acc, num) => acc + num, 0); // Sum of unique  
  numbers  
  const totalSum = (n * (n + 1)) / 2; // Sum of first n natural numbers  
  return totalSum - uniqueSum;  
}  
  
console.log(findMissingWithDuplicates([1, 2, 2, 4, 5], 5)); // Output: 3
```

### **Explanation:**

#### **1. Handle Duplicates:**

Use new Set() to remove duplicates from the array.

#### **2. Calculate the Total Sum:**

Use  $(n * (n + 1)) / 2$  to get the sum of numbers from 1 to n.

### 3. Find the Missing Number:

Subtract the sum of unique numbers from the total sum to get the missing number.

### 4. Output:

For the array [1, 2, 2, 4, 5], the missing number is 3.

## Reel 24:

### Find the Duplicates in an Array

**1. Problem: You are given an array of integers. Write a program to find all the duplicate elements in the array.**

#### Input and Output:

Input: [4, 3, 2, 7, 8, 2, 3, 1];  
Output: [2, 3];

Input: [1, 2, 3, 4, 5];  
Output: [];

Input: [1, 1, 1, 2, 2, 3];  
Output: [1, 2];

#### Solution:

```
function findDuplicates(arr) {  
    const seen = new Set();  
    const duplicates = new Set();  
  
    for (let num of arr) {  
        if (seen.has(num)) {  
            duplicates.add(num);  
        } else {  
            seen.add(num);  
        }  
    }  
  
    return [...duplicates];  
}  
  
console.log(findDuplicates([4, 3, 2, 7, 8, 2, 3, 1])); // Output: [2, 3]  
console.log(findDuplicates([1, 2, 3, 4, 5])); // Output: []  
console.log(findDuplicates([1, 1, 1, 2, 2, 3])); // Output: [1, 2]
```

## Explanation:

### 1. Initialization:

- o Use a Set to keep track of seen numbers (seen).
- o Use another Set to store duplicate elements (duplicates).

### 2. Iterate Through the Array:

- o For each element:
  - Check if it exists in the seen set.
  - If yes, it's a duplicate; add it to the duplicates set.
  - Otherwise, add it to the seen set.

### 3. Output the Result:

- o Convert the duplicates set into an array using [...duplicates].

### 4. Edge Cases:

- o If the array has no duplicates, return an empty array ([]).

## Find All Pairs with a Given Sum

**2. Problem:** You are given an array of integers and a target sum. Write a program to find all unique pairs of numbers in the array that add up to the target sum.

### Input and Output:

Input: (arr = [1, 2, 3, 4, 5]), (target = 6);

Output: [  
[1, 5],  
[2, 4],  
];

Input: (arr = [2, 4, 3, 7, 8]), (target = 10);

Output: [  
[2, 8],  
[3, 7],  
];

Input: (arr = [1, 1, 1]), (target = 2);

Output: [[1, 1]];

## Solution:

```
function findPairs(arr, target) {  
    const seen = new Set();  
    const pairs = [];  
  
    for (let num of arr) {  
        const complement = target - num;  
  
        if (seen.has(complement)) {
```

```

        pairs.push([complement, num]);
    }

    seen.add(num);
}

return pairs;
}

console.log(findPairs([1, 2, 3, 4, 5], 6)); // Output: [[1, 5], [2, 4]]
console.log(findPairs([2, 4, 3, 7, 8], 10)); // Output: [[2, 8], [3, 7]]
console.log(findPairs([1, 1, 1], 2)); // Output: [[1, 1]]

```

### Explanation:

1. **Understand the Complement:**
  - o For a target sum target and current number num, the complement is target - num.
2. **Iterate Through the Array:**
  - o Check if the complement exists in the Set of seen numbers.
  - o If yes, add the pair [complement, num] to the pairs array.
3. **Update the Seen Set:**
  - o Add the current number to the Set.
4. **Return the Pairs:**
  - o Return all pairs that sum to the target value.
5. **Edge Cases:**
  - o If there are no pairs, return an empty array ([]).

## Find the Majority Element

**3. Problem:** You are given an array of integers. Write a program to find the majority element (the element that appears more than  $n / 2$  times). If no such element exists, return null.

### Input and Output:

Input: [3, 3, 4, 2, 4, 4, 2, 4, 4];  
Output: 4;

Input: [1, 2, 3, 4];  
Output: null;

### Solution:

```

function findMajorityElement(arr) {
    const countMap = {};

    for (let num of arr) {
        countMap[num] = (countMap[num] || 0) + 1;
    }
}
```

```

if (countMap[num] > arr.length / 2) {
    return num;
}
}

return null;
}

console.log(findMajorityElement([3, 3, 4, 2, 4, 4, 2, 4, 4])); // Output: 4
console.log(findMajorityElement([1, 2, 3, 4])); // Output: null

```

### Explanation:

1. **Frequency Map:**
  - o Use an object (countMap) to count the frequency of each number.
2. **Iterate Through the Array:**
  - o Increment the count for each number in countMap.
3. **Check the Majority Condition:**
  - o If the count of a number exceeds  $n / 2$ , return that number.
4. **Output the Result:**
  - o If no majority element is found, return null.
5. **Edge Cases:**
  - o For an array with no majority element, the result is null.

### Rotate an Array to the Right

*Problem: Rotate an array to the right by k steps.*

Input and Output:

Input: (arr = [1, 2, 3, 4, 5]), (k = 2);  
 Output: [4, 5, 1, 2, 3];

Input: (arr = [7, 8, 9]), (k = 1);  
 Output: [9, 7, 8];

### Solution:

```

function rotateArray(arr, k) {
    k = k % arr.length; // Handle cases where k > arr.length
    return [...arr.slice(-k), ...arr.slice(0, -k)];
}

console.log(rotateArray([1, 2, 3, 4, 5], 2)); // Output: [4, 5, 1, 2, 3]
console.log(rotateArray([7, 8, 9], 1)); // Output: [9, 7, 8]

```

### **Explanation:**

1. **Handle Large k:**
  - o Use  $k = k \% \text{arr.length}$  to handle cases where  $k$  is greater than the array length.
2. **Split the Array:**
  - o Use `slice()` to extract the last  $k$  elements (`arr.slice(-k)`).
  - o Use `slice()` to extract the remaining elements (`arr.slice(0, -k)`).
3. **Combine the Parts:**
  - o Combine the two parts using the spread operator (...).
4. **Output:**
  - o For  $[1, 2, 3, 4, 5]$  and  $k = 2$ , the result is  $[4, 5, 1, 2, 3]$ .
5. **Edge Cases:**
  - o If  $k = 0$  or  $k$  is a multiple of the array length, the array remains unchanged.

### **Reel 25:**

**Question 1:** You are given an array of integers. Write a program to find the **maximum product of two distinct elements** in the array.

### **Examples:**

1. Input:  $[1, 2, 3, 4, 5]$   
Output: 20  
(Explanation: The maximum product is  $4 * 5 = 20$ )
2. Input:  $[7, -2, 3, -5, 10]$   
Output: 35  
(Explanation: The maximum product is  $-5 * -7 = 35$ )
3. Input:  $[1, 1, 1]$   
Output: 1  
(Explanation: The only pair possible is  $1 * 1 = 1$ )

### **Code:**

```
function maxProductOfTwo(arr) {  
    if (arr.length < 2) {  
        return null; // Return null if there aren't at least 2 elements  
    }  
  
    // Initialize variables  
    let max1 = -Infinity; // Largest number  
    let max2 = -Infinity; // Second largest number  
    let min1 = Infinity; // Smallest number  
    let min2 = Infinity; // Second smallest number  
  
    // Iterate through the array to find the top 2 largest and 2 smallest numbers  
    for (let num of arr) {  
        // Update max1 and max2  
        if (num > max1) {  
            max2 = max1;  
            max1 = num;  
        } else if (num < min1) {  
            min2 = min1;  
            min1 = num;  
        } else if (num > max2) {  
            max2 = num;  
        } else if (num < min2) {  
            min2 = num;  
        }  
    }  
  
    return max1 * max2;  
}  
  
// Test cases  
console.log(maxProductOfTwo([1, 2, 3, 4, 5])); // 20  
console.log(maxProductOfTwo([-10, 1, 3, 5, 7, 9])); // 63  
console.log(maxProductOfTwo([-5, -4, -3, -2, -1, -10])); // 20  
console.log(maxProductOfTwo([1, 1, 1, 1, 1])); // 1  
console.log(maxProductOfTwo([])); // null
```

```

max1 = num;
} else if (num > max2) {
  max2 = num;
}

// Update min1 and min2
if (num < min1) {
  min2 = min1;
  min1 = num;
} else if (num < min2) {
  min2 = num;
}
}

// Calculate the maximum product
const product1 = max1 * max2; // Product of two largest numbers
const product2 = min1 * min2; // Product of two smallest numbers (handles negatives)

return Math.max(product1, product2);
}

// Test cases
console.log(maxProductOfTwo([1, 2, 3, 4, 5])); // Output: 20
console.log(maxProductOfTwo([7, -2, 3, -5, 10])); // Output: 35
console.log(maxProductOfTwo([1, 1, 1])); // Output: 1
console.log(maxProductOfTwo([2])); // Output: null

```

## Explanation:

1. **Handle edge cases:**
  - o If the array has less than two elements, return null since a product can't be calculated.
2. **Find the largest and smallest values:**
  - o Use variables max1 and max2 to store the two largest numbers in the array.
  - o Use variables min1 and min2 to store the two smallest numbers in the array (important for cases involving negative numbers).
3. **Iterate through the array:**
  - o Compare each number with max1 and max2 to update the largest numbers.
  - o Similarly, compare each number with min1 and min2 to update the smallest numbers.
4. **Calculate the maximum product:**
  - o Compute the product of the two largest numbers (max1 \* max2).
  - o Compute the product of the two smallest numbers (min1 \* min2) to handle cases where multiplying two negative numbers results in a larger positive product.
  - o Return the maximum of these two products.

**Question 2: Write a program to find the maximum product of three distinct elements in an array.**

**Examples:**

Input: [1, 2, 3, 4, 5]

Output: 60

(Explanation: The maximum product is  $3 * 4 * 5 = 60$ )

Input: [-10, -20, 5, 1, 3]

Output: 1000

(Explanation: The maximum product is  $-10 * -20 * 5 = 1000$ )

**Code:**

```
function maxProductOfThree(arr) {  
    if (arr.length < 3) {  
        return null; // Not enough elements  
    }  
  
    arr.sort((a, b) => a - b); // Sort the array in ascending order  
  
    const n = arr.length;  
    const product1 = arr[n - 1] * arr[n - 2] * arr[n - 3]; // Product of the three largest numbers  
    const product2 = arr[0] * arr[1] * arr[n - 1]; // Product of two smallest and the largest number  
  
    return Math.max(product1, product2);  
}  
  
// Test cases  
console.log(maxProductOfThree([1, 2, 3, 4, 5])); // Output: 60  
console.log(maxProductOfThree([-10, -20, 5, 1, 3])); // Output: 1000  
console.log(maxProductOfThree([1, 1, 1])); // Output: 1
```

**Question 3: Write a program to find the maximum product of any two adjacent elements in the array.**

**Examples:**

Input: [3, 6, -2, -5, 7, 3]

Output: 21  
(Explanation: The maximum product is  $7 * 3 = 21$ )  
Input: [-1, -3, 10, 0, 5]  
Output: 30  
(Explanation: The maximum product is  $-3 * 10 = 30$ )

**Code:**

```
function maxAdjacentProduct(arr) {  
    if (arr.length < 2) {  
        return null; // Not enough elements  
    }  
  
    let maxProduct = -Infinity;  
  
    for (let i = 0; i < arr.length - 1; i++) {  
        const product = arr[i] * arr[i + 1];  
        maxProduct = Math.max(maxProduct, product);  
    }  
  
    return maxProduct;  
}  
  
// Test cases  
console.log(maxAdjacentProduct([3, 6, -2, -5, 7, 3])); // Output: 21  
console.log(maxAdjacentProduct([-1, -3, 10, 0, 5])); // Output: 30  
console.log(maxAdjacentProduct([5, 5, 5])); // Output: 25
```

**Explanation:**

- The function iterates through the array and calculates the product of every pair of adjacent elements.
- The maximum product is tracked and returned.

Question 4: Find the maximum product of k distinct elements in an array, where k is given.

**Examples:**

Input: arr = [1, 10, 2, 6, 5, 3], k = 3  
Output: 300  
(Explanation: The maximum product is  $10 * 6 * 5 = 300$ )  
Input: arr = [-1, -2, -3, -4, -5], k = 2  
Output: 20  
(Explanation: The maximum product is  $-4 * -5 = 20$ )

**Code:**

```
function maxProductOfK(arr, k) {
```

```

if (arr.length < k) {
    return null; // Not enough elements
}

arr.sort((a, b) => Math.abs(b) - Math.abs(a)); // Sort by absolute values in descending order

let product = 1;
for (let i = 0; i < k; i++) {
    product *= arr[i];
}

return product;
}

// Test cases
console.log(maxProductOfK([1, 10, 2, 6, 5, 3], 3)); // Output: 300
console.log(maxProductOfK([-1, -2, -3, -4, -5], 2)); // Output: 20
console.log(maxProductOfK([1, 1, 1], 2)); // Output: 1

```

### Explanation:

- The array is sorted by the absolute value of each element (to handle negatives).
- The top k elements are multiplied to find the maximum product.

Question 5: **Write a program to find the minimum product of two distinct elements in the array.**

### Examples:

```

Input: [1, 2, 3, 4, 5]
Output: 2
(Explanation: The minimum product is 1 * 2 = 2)
Input: [-1, -3, 10, 0, 5]
Output: -30
(Explanation: The minimum product is -3 * 10 = -30)

```

### Code:

```

function minProductOfTwo(arr) {
    if (arr.length < 2) {
        return null; // Not enough elements
    }

    arr.sort((a, b) => a - b); // Sort the array in ascending order
    const n = arr.length;

    // Two possible minimum products

```

```

const product1 = arr[0] * arr[1]; // Two smallest numbers
const product2 = arr[n - 2] * arr[n - 1]; // Two largest numbers (if negative)

return Math.min(product1, product2);
}

// Test cases
console.log(minProductOfTwo([1, 2, 3, 4, 5])); // Output: 2
console.log(minProductOfTwo([-1, -3, 10, 0, 5])); // Output: -30
console.log(minProductOfTwo([-10, -20, -30])); // Output: -600

```

### **Explanation:**

- The array is sorted to find the smallest two elements.
- Both the smallest and largest pairs are checked (to handle negative products).
- The minimum product is returned.

### **Reel 26:**

**Question 1: You are given a string containing letters, digits, and special characters. Write a program to reverse only the letters in the string while keeping all other characters in the same position.**

### **Examples:**

1. Input: "a1b!c"  
Output: "c1b!a"  
(Explanation: Only the letters 'a', 'b', and 'c' are reversed, while 1 and ! remain in place.)

2. Input: "h@el#lo!"  
Output: "o@ll#eh!"  
(Explanation: 'h', 'e', 'l', 'o' are reversed, while @, #, and ! stay in their positions.)

3. Input: "12345"  
Output: "12345"  
(Explanation: No letters to reverse, so the string remains the same.)

### **Code:**

```

function reverseOnlyLetters(s) {
  let letters = [] // Array to store only the letters

  // Extract letters from the string
  for (let char of s) {
    if (/^[a-zA-Z]$/.test(char)) {
      // Check if character is a letter
      letters.push(char);
    }
  }
}

```

```

}

let result = ""; // To store the final output
let letterIndex = letters.length - 1; // Start from the last letter

// Reconstruct the string
for (let char of s) {
  if (/[^a-zA-Z]/.test(char)) {
    // If it's a letter, replace it with the last letter in stored array
    result += letters[letterIndex];
    letterIndex--; // Move to the previous letter in the stored array
  } else {
    result += char; // Keep non-letter characters in place
  }
}

return result;
}

// Test cases
console.log(reverseOnlyLetters("a1b!c")); // Output: "c1b!a"
console.log(reverseOnlyLetters("h@el#lo!")); // Output: "o@ll#eh!"
console.log(reverseOnlyLetters("12345")); // Output: "12345"
console.log(reverseOnlyLetters("C0d!ng_123")); // Output: "g0n!dC_123"

```

## Explanation:

### 1. Extract all letters

- The function iterates through the string and collects only the **letters** in an array (letters).
- Example: "h@el#lo!" → letters = ['h', 'e', 'l', 'l', 'o']

### 2. Reconstruct the string

- It iterates through the original string again.
- If a character is a **letter**, it replaces it with the last stored letter (from letters).
- If it's a **non-letter** (digit, special character), it remains unchanged.
- Example:
  - 'h' → 'o'
  - 'e' → 'l'
  - 'l' → 'l'
  - 'l' → 'e'
  - 'o' → 'h'
  - @, #, ! remain unchanged.

### 3. Return the modified string

## Question 2: Find the Longest Word in a Sentence

### Problem:

You are given a sentence. Write a program to find the longest word in the sentence.

Examples:

#### *Input:*

```
"The quick brown fox";
```

#### Code:

```
function longestWord(sentence) {  
    let words = sentence.split(" ");  
    let longest = "";  
  
    for (let word of words) {  
        if (word.length > longest.length) {  
            longest = word;  
        }  
    }  
  
    return longest;  
}  
  
// Test cases  
console.log(longestWord("The quick brown fox")); // Output: "quick"  
console.log(longestWord("JavaScript is awesome")); // Output: "JavaScript"  
console.log(longestWord("Hi")); // Output: "Hi"
```

#### Explanation:

1. Split the sentence into words using `.split(" ")`.
2. Iterate through the words and track the longest one.
3. Return the longest word found.

## Question 3: You are given a string. Write a program to count the occurrences of each character in the string.

#### *Input:*

```
"banana";
```

#### Code:

```
function countCharacterOccurrences(str) {  
    let frequency = {};
```

```

for (let char of str) {
    frequency[char] = (frequency[char] || 0) + 1;
}

return frequency;
}

// Test cases
console.log(countCharacterOccurrences("banana")); // Output: { b: 1, a: 3, n: 2 }
console.log(countCharacterOccurrences("hello")); // Output: { h: 1, e: 1, l: 2, o: 1 }
console.log(countCharacterOccurrences("apple")); // Output: { a: 1, p: 2, l: 1, e: 1 }

```

### **Explanation:**

1. Create an empty object frequency to store character counts.
2. Iterate through the string, updating the count of each character.
3. Return the final object containing character frequencies.

### **Reel 27:**

#### Remove Vowels from a String

##### **1. Problem: Write a program to remove all vowels (a, e, i, o, u) from a given string.**

##### *Input:*

"JavaScript is fun";

##### **Output:**

"JvScrt s fn";

##### **Solution:**

```

function removeVowels(str) {
    return str.replace(/[aeiouAEIOU]/g, ""); // Remove vowels using regex
}

// Test cases
console.log(removeVowels("JavaScript is fun")); // Output: "JvScrt s fn"
console.log(removeVowels("Hello World")); // Output: "Hll Wrld"
console.log(removeVowels("I love programming")); // Output: " lv prgrmmng"
console.log(removeVowels("AEIOU are vowels")); // Output: " r vwls"

```

Explanation:

1. Use `replace()` with a **regular expression** (`/[aeiouAEIOU]/g`) to match all vowels.
2. Replace them with an empty string (""), effectively removing them.

### Find the Longest Word in a Sentence

**2. Problem: You are given a sentence as a string. Write a program to find the longest word in the sentence.**

*Input:*

```
"JavaScript makes coding enjoyable";
```

**Output:**

```
"JavaScript";
```

**Solution:**

```
function findLongestWord(sentence) {  
    let words = sentence.split(" ");  
    let longest = "";  
  
    for (let word of words) {  
        if (word.length > longest.length) {  
            longest = word;  
        }  
    }  
  
    return longest;  
}  
  
// Test cases  
console.log(findLongestWord("JavaScript makes coding enjoyable")); // Output: "JavaScript"  
console.log(findLongestWord("I love programming")); // Output: "programming"  
console.log(findLongestWord("The quick brown fox jumps over the lazy dog")); // Output:  
"jumps"  
console.log(findLongestWord("Hello world!")); // Output: "Hello"
```

Explanation:

1. **Split** the string into an array of words.
2. **Iterate** through the words and compare their lengths.
3. **Return** the word with the maximum length.

## Capitalize the First Letter of Each Word

**3. Problem:** Write a program to capitalize the first letter of each word in a given sentence.

*Input:*

```
"javascript is powerful";
```

**Output:**

```
"Javascript Is Powerful";
```

**Solution:**

```
function capitalizeWords(sentence) {
    return sentence
        .split(" ") // Split the sentence into words
        .map((word) => word.charAt(0).toUpperCase() + word.slice(1)) // Capitalize first letter
        .join(" "); // Join words back into a sentence
}

// Test cases
console.log(capitalizeWords("javascript is powerful")); // Output: "Javascript Is Powerful"
console.log(capitalizeWords("hello world")); // Output: "Hello World"
console.log(capitalizeWords("i love coding")); // Output: "I Love Coding"
console.log(capitalizeWords("a quick brown fox")); // Output: "A Quick Brown Fox"
```

Explanation:

1. **Split** the string into words.
2. **Capitalize** the first letter using `.charAt(0).toUpperCase() + word.slice(1)`.
3. **Join** the modified words back into a sentence.

## Reverse Letters in Each Word

**4. Problem:** You are given a string with words separated by spaces. Write a program to reverse the letters in each word while keeping the word order the same.

**Examples:**

**Input:**

```
"tpircSavaJ si emosewa"
```

**Solution:**

```

function reverseLettersInWords(sentence) {
  return sentence
    .split(" ") // Split sentence into words
    .map((word) => word.split("").reverse().join("")) // Reverse each word
    .join(" "); // Join words back into a sentence
}

// Test cases
console.log(reverseLettersInWords("JavaScript is awesome")); // Output: "tpircSavaJ si emosewa"
console.log(reverseLettersInWords("I love coding")); // Output: "I evol gnidoc"
console.log(reverseLettersInWords("Hello World")); // Output: "olleH dlroW"
console.log(reverseLettersInWords("SingleWord")); // Output: "droWelgniS"

```

Explanation:

1. **Split** the string into an array of words.
2. **Reverse** the letters of each word using `.split("")``.reverse()``.join("")`.
3. **Join** the modified words back into a string.

## Reel 28:

### Find the most frequent character

**Problem:** You are given a string. Write a program to find the most frequently occurring character in the string. If multiple characters have the same highest frequency, return the first one encountered.

```

Input: "javascript"
Output: "a"
(Explanation: The letter "a" appears twice, which is the highest frequency.)

Input: "apple"
Output: "p"
(Explanation: The letter "p" appears twice, which is the most frequent character.)

Input: "hello world"
Output: "l"
(Explanation: The letter "l" appears three times, which is the highest frequency.)

Input: "abcd"
Output: "a"
(Explanation: All characters appear only once, so we return the first character encountered.)

```

### Solution:

```

function mostFrequentCharacter(str) {
  let frequency = {} // Object to store character counts
  let maxChar = ""; // Stores the character with the highest frequency
  let maxCount = 0; // Stores the highest frequency count

```

```

for (let char of str) {
  if (char !== " ") {
    // Ignore spaces
    frequency[char] = (frequency[char] || 0) + 1; // Count each character

    // Update maxChar if the current character has a higher frequency
    if (frequency[char] > maxCount) {
      maxCount = frequency[char];
      maxChar = char;
    }
  }
}

return maxChar;
}

// Test cases
console.log(mostFrequentCharacter("javascript")); // Output: "a"
console.log(mostFrequentCharacter("apple")); // Output: "p"
console.log(mostFrequentCharacter("hello world")); // Output: "l"
console.log(mostFrequentCharacter("abcd")); // Output: "a"

```

### **Explanation:**

1. **Create an empty object** `frequency` to store character counts.
2. **Loop through the string**, counting occurrences of each character.
  - o If a character appears more times than `maxCount`, update `maxCount` and `maxChar`.
3. **Ignore spaces (" ")** to focus only on letters.
4. **Return the character with the highest frequency**.
  - o If multiple characters have the same count, the first encountered is returned.

**O(n)** → We iterate through the string **once** to count characters and **once** to find the max.

### **Find the First Non-Repeating Character**

**Problem:** You are given a string. Write a program to find the first non-repeating character in the string. If all characters repeat, return null.

Examples:

```

Input:
"javascript"
Output:
"j"
(Explanation: "j" appears only once, and it's the first unique character.)
Input:
"apple"

```

```

Output:
"a"
(Explanation: "a" appears once before other unique characters.)

Input:
"aabbcc"~
Output:
null
(Explanation: All characters repeat, so we return null.)

```

### Solution:

```

function firstNonRepeatingCharacter(str) {
    let frequency = {};
    // Count occurrences of each character
    for (let char of str) {
        frequency[char] = (frequency[char] || 0) + 1;
    }
    // Find the first non-repeating character
    for (let char of str) {
        if (frequency[char] === 1) {
            return char;
        }
    }
    return null; // If all characters repeat
}

// Test cases
console.log(firstNonRepeatingCharacter("javascript")); // Output: "j"
console.log(firstNonRepeatingCharacter("apple")); // Output: "a"
console.log(firstNonRepeatingCharacter("aabbcc")); // Output: null

```

### Explanation:

1. Create a frequency object to count occurrences of each character.
2. Iterate through the string again and return the first character that appears only once.
3. If no unique character exists, return null.

### Check If Two Strings Are Anagrams

**Problem:** You are given two strings. Write a program to check if they are anagrams (contain the same letters in a different order).

Examples:

```
Input:  
"listen", "silent"  
Output:  
true  
(Explanation: Both words contain the same letters, rearranged.)  
Input:  
"hello", "world"  
Output:  
false  
(Explanation: Different letters, so not anagrams.)
```

### Solution:

```
function areAnagrams(str1, str2) {  
    // Sort both strings and compare  
    return str1.split("").sort().join("") === str2.split("").sort().join("");  
}  
  
// Test cases  
console.log(areAnagrams("listen", "silent")); // Output: true  
console.log(areAnagrams("hello", "world")); // Output: false  
console.log(areAnagrams("triangle", "integral")); // Output: true
```

### Explanation:

1. Convert both strings into arrays using `.split("")`.
2. Sort the arrays using `.sort()`.
3. Join the sorted arrays back into strings using `.join("")` and compare them.

## Find the Longest Word in a Sentence

**Problem:** You are given a sentence as a string. Write a program to find the longest word in the sentence.

Examples:

```
Input:  
"The quick brown fox jumps"  
Output:  
"jumps"  
(Explanation: "jumps" is the longest word with 5 letters.)  
Input:  
"I love JavaScript"  
Output:  
"JavaScript"  
(Explanation: "JavaScript" is the longest word with 10 letters.)
```

### Solution:

```

function longestWord(sentence) {
  let words = sentence.split(" ");
  let longest = "";

  for (let word of words) {
    if (word.length > longest.length) {
      longest = word;
    }
  }

  return longest;
}

// Test cases
console.log(longestWord("The quick brown fox jumps")); // Output: "jumps"
console.log(longestWord("I love JavaScript")); // Output: "JavaScript"
console.log(longestWord("Coding is fun")); // Output: "Coding"

```

### **Explanation:**

1. **Split** the sentence into an array of words.
2. **Iterate** through each word, keeping track of the longest one.
3. **Return** the longest word found.

### **Check If a String Contains Only Unique Characters**

**Problem:** Write a program to check if a given string contains only unique characters.

Examples:

```

Input:
"abcdef"
Output:
true
(Explanation: All characters are unique.)

Input:
"hello"
Output:
false
(Explanation: "l" appears twice.)

Input:
"world"
Output:
true
(Explanation: All characters are unique.)

```

### Solution:

```
function hasUniqueCharacters(str) {  
    let seen = new Set();  
  
    for (let char of str) {  
        if (seen.has(char)) {  
            return false; // If character is repeated, return false  
        }  
        seen.add(char);  
    }  
  
    return true;  
}  
  
// Test cases  
console.log(hasUniqueCharacters("abcdef")); // Output: true  
console.log(hasUniqueCharacters("hello")); // Output: false  
console.log(hasUniqueCharacters("world")); // Output: true
```

### Explanation:

1. **Use a Set** to track characters seen before.
2. **Iterate** through the string:
  - o If a character is already in the Set, return false.
  - o Otherwise, add it to the Set.
3. If no duplicates are found, return true.