FRONT END INTERVIEW QUESTIONS

1. **Differences between HTML and HTML5:**
   - **HTML**: Older version, mainly defines structure of webpages using basic tags (e.g., ,).
   - **HTML5**: Newer version, includes new semantic tags (e.g., <article>, <section>), APIs (e.g., geolocation, local storage), and multimedia support (e.g., <audio>, <video>).
2. **Differences between Local Storage and Session Storage:**
   - **Local Storage**: Stores data with no expiration date; persists even after the browser is closed.
   - **Session Storage**: Stores data for the duration of the page session; data is cleared when the tab or browser is closed.
3. **Multimedia tags in HTML5**:
   - <audio>: Embeds audio content.
   - <video>: Embeds video content.
   - <track>: Provides subtitles or captions for videos.
4. **Structure of a webpage**: A basic webpage structure includes:
   - <html>: Root element.
   - <head>: Contains metadata (e.g., title, links to stylesheets).
   - <body>: Contains the actual content of the webpage.
5. **Meta tag**: The <meta> tag provides metadata about the HTML document, like character encoding or author information.
6. **<meta name="viewport" content="width=device-width, initial-scale=1.0">**: This tag controls the layout on mobile devices by setting the width of the page to the device's width and scaling the content to an initial zoom level.
7. **<iframe> tag**: The <iframe> tag is used to embed another HTML document within the current document.
8. **Attributes**: Attributes provide additional information about an element, like src, href, or class.
9. **Execution flow of HTML, CSS, and JS**:
   - **HTML**: The browser parses the HTML structure.
   - **CSS**: The browser applies the styles to the HTML elements.
   - **JS**: JavaScript runs, modifying the DOM and responding to user actions.
10. **Anchor element**: The <a> element defines a hyperlink, allowing navigation to other pages or resources.
11. ** tag**: The tag is an inline container for styling or grouping text within other elements.
12. **Ways to apply styles**:

- Inline styles (style attribute).
- Internal styles (<style> tag in the <head>).
- External styles (linked through <link> tag).

13. **Differences between internal styles and external styles**:
    - **Internal**: Styles defined within the <style> tag in the HTML document.
    - **External**: Styles defined in a separate .css file linked to the HTML.
14. **CSS Specificity**: CSS specificity determines which rule applies when multiple rules match an element. It's calculated based on inline styles, IDs, classes, and element selectors.
15. **Selectors**: Selectors are patterns used to target elements in the DOM (e.g., id, class, element).
16. **Differences between ID and Class selectors**:
    - **ID**: Unique identifier for an element; prefixed with #.
    - **Class**: Can be used for multiple elements; prefixed with .
17. **Differences between rem and em dimensions**:
    - **rem**: Relative to the root font size.
    - **em**: Relative to the font size of the parent element.
18. **All dimensions in CSS**:
    - **px**: Pixels.
    - **em**: Relative to the parent element's font size.
    - **rem**: Relative to the root font size.
    - **%**: Relative to the parent element.
    - **vw/vh**: Viewport width/height.
19. **Breakpoints/Media queries**: Media queries allow applying styles based on the device's characteristics (e.g., screen width).
20. **Differences between Flex and Grid**:
    - **Flexbox**: 1D layout system (either row or column).
    - **Grid**: 2D layout system (both rows and columns).
21. **Media queries**: Media queries apply different styles based on the device's properties (like screen width).
22. **Positions in CSS**:
    - static, relative, absolute, fixed, sticky.
23. **Differences between programming and scripting languages**:
    - **Programming languages**: Can create standalone applications (e.g., C, Java).

- **Scripting languages**: Typically used for web development to manipulate the DOM or automate tasks (e.g., JavaScript, Python).

24. **Differences between var and let**:
    - **var**: Function-scoped; allows redeclaration.
    - **let**: Block-scoped; cannot be redeclared in the same scope.
25. **Const Keyword**: const defines a constant variable, whose value cannot be reassigned after initialization.
26. **Variable Hoisting**: Variable hoisting refers to JavaScript's behavior of moving variable and function declarations to the top of their scope during execution.
27. **Global Pollution Issue**: Global pollution occurs when too many global variables are created, which can lead to name conflicts and unpredictable behavior.
28. **Functions**: Functions are blocks of code designed to perform a particular task when called.
29. **Differences between normal functions and arrow functions**:
    - **Normal**: Defined with the function keyword, have their own this.
    - **Arrow**: Shorter syntax, inherit this from the surrounding context.
30. **Callbacks**: Functions passed as arguments to other functions, executed after the completion of a task.
31. **Callback Hell**: Nested callbacks that can make the code hard to read and maintain.
32. **Differences between callbacks and promises**:
    - **Callbacks**: Handle async results but can lead to callback hell.
    - **Promises**: Represent a value that may not be available yet, allowing cleaner handling of async operations.
33. **Promises**: Promises represent the eventual completion (or failure) of an asynchronous operation.
34. **Async and Await keywords**: async defines a function that returns a promise, and await pauses execution until the promise is resolved.
35. **Synchronous and Asynchronous**:
    - **Synchronous**: Tasks are executed sequentially.
    - **Asynchronous**: Tasks are executed concurrently, allowing for non-blocking code.
36. **Prototype**: A prototype is an object that provides shared properties and methods to other objects.
37. **Prototype Chaining**: When a property or method is not found in an object, JavaScript looks up the prototype chain.
38. **Prototype and __proto__ properties**:
    - **Prototype**: Defines shared properties/methods.

- **__proto__**: Refers to an object's prototype.

39. **IIFE Functions and Advantages**: Immediately Invoked Function Expressions (IIFE) are functions that run as soon as they are defined, preventing global scope pollution.
40. **Asynchronous calls with generator functions**: Generators are functions that can pause and resume, useful for managing async operations.
41. **Yield keyword**: yield pauses the execution of a generator function and can return a value.
42. **Differences between Promise.all(), Promise.race(), Promise.allSettled(), Promise.any()**:

- **all()**: Resolves when all promises resolve.
- **race()**: Resolves when the first promise resolves or rejects.
- **allSettled()**: Resolves when all promises are settled (either resolved or rejected).
- **any()**: Resolves when any one of the promises resolves.

43. **JSON**: JSON (JavaScript Object Notation) is a lightweight data-interchange format.
44. **Object functions**:

- **Object.assign()**: Copies properties from one object to another.
- **Object.create()**: Creates a new object with the specified prototype.
- **Object.entries()**: Returns an array of key-value pairs.
- **Object.fromEntries()**: Converts key-value pairs into an object.

45. **Closures and private members**: A closure is a function that remembers its lexical scope even when the function is executed outside that scope, allowing the creation of private members.
46. **Reusability with higher-order functions**: Higher-order functions accept other functions as arguments or return them, allowing reuse of code.
47. **Map(), filter(), reduce()**:

- **map()**: Transforms each element in an array.
- **filter()**: Filters elements based on a condition.
- **reduce()**: Reduces an array to a single value based on a callback.

48. **Differences between includes() and find()**:

- **includes()**: Checks if an array contains a specific value.
- **find()**: Returns the first element that satisfies a condition.

49. **Differences between splice() and slice()**:

- **splice()**: Modifies the array by adding/removing elements.
- **slice()**: Returns a shallow copy of a portion of the array.

50. **Differences between some() and every()**:

- **some()**: Returns true if at least one element satisfies the condition.
- **every()**: Returns true if all elements satisfy the condition.

51. **findIndex() vs indexOf()**

- **findIndex()**: Returns the index of the first element in an array that satisfies a given condition. It uses a callback function that tests each element.
- const arr = [5, 12, 8, 130, 44];
- const index = arr.findIndex(num => num > 10); // returns 1 (index of 12)
- **indexOf()**: Returns the index of the first occurrence of a specified value in an array, or -1 if not found.
- const arr = [5, 12, 8, 130, 44];
- const index = arr.indexOf(12); // returns 1

52. **toString() vs join()**

- **toString()**: Converts the array to a string where each element is separated by commas.
- const arr = [1, 2, 3];
- console.log(arr.toString()); // "1,2,3"
- **join()**: Joins all elements of an array into a string, but you can specify the separator.
- const arr = [1, 2, 3];
- console.log(arr.join('-')); // "1-2-3"

53. **setTimeout(), setInterval(), clearInterval(), clearTimeout()**

- **setTimeout()**: Executes a function once after a specified delay.
- setTimeout(() => console.log("Hello"), 2000); // prints "Hello" after 2 seconds
- **setInterval()**: Executes a function repeatedly at specified intervals.
- setInterval(() => console.log("Hello"), 2000); // prints "Hello" every 2 seconds
- **clearInterval()**: Clears an interval that was set by setInterval().
- let intervalId = setInterval(() => console.log("Hello"), 2000);
- clearInterval(intervalId); // stops the interval
- **clearTimeout()**: Clears a timeout that was set by setTimeout().
- let timeoutId = setTimeout(() => console.log("Hello"), 2000);
- clearTimeout(timeoutId); // stops the timeout

54. **Regular Expressions for Validation**

- **Email Validation**: /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/

- **Password Validation**: /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}$/ (at least 8 characters, one letter, one number)
- **Mobile Validation**: /^\d{10}$/
- **Username Validation**: /^[A-Za-z0-9_]{5,15}$/ (5 to 15 characters, alphanumeric and underscores)

55. **Event Bubbling and Event Capturing**

- **Event Bubbling**: Events bubble up from the target element to the root element.
- **Event Capturing**: Events are captured starting from the root element down to the target element.

56. **Preventing Event Propagation**

- Use event.stopPropagation() to prevent the event from propagating to parent elements.

```
element.addEventListener('click', function(event) {
  event.stopPropagation();
});
```

57. **Currying**

- **Currying**: The process of transforming a function that takes multiple arguments into a series of functions that each take a single argument.

```
function multiply(a) {
  return function(b) {
  return a * b;
  };
}
const multiplyBy2 = multiply(2);
console.log(multiplyBy2(5)); // 10
```

58. **Primitive vs Non-Primitive Data Types**

- **Primitive Data Types**: Immutable and passed by value (e.g., String, Number, Boolean, Null, Undefined).
- **Non-Primitive Data Types**: Mutable and passed by reference (e.g., Object, Array, Function).

59. **ES6 Features**

    - Let and Const
    - Arrow Functions
    - Template Literals
    - Default Parameters
    - Rest and Spread Operators

- Destructuring Assignment
- Classes
- Promises
- Modules (import/export)
- Let/Const Block Scoping

60. **Debouncing vs Throttling**

- **Debouncing**: Delays the execution of a function until a certain time has passed since the last event.
- **Throttling**: Ensures a function is called at most once in a specified period.

61. **this Keyword**

- Refers to the context in which a function is called. It could be the global object, the object itself, or undefined in strict mode.

62. **Lexical Scope**

- Refers to the scope that is determined by the location of the variables and functions in the source code. Inner functions have access to variables in their outer functions.

63. **Temporal Dead Zone (TDZ)**

- The period between the entering of the scope and the variable's initialization where accessing the variable will throw a ReferenceError.

64. **call(), apply(), bind()**

- **call()**: Invokes a function with a specified this value and arguments.
- **apply()**: Similar to call(), but accepts an array of arguments.
- **bind()**: Returns a new function that, when called, has its this value set to the provided value.

65. **Making API Calls with JavaScript**

- Use fetch() or XMLHttpRequest() to make API calls.
- fetch('https://api.example.com')
- .then(response => response.json())
- .then(data => console.log(data));

66. **ReactJS Features**

- Virtual DOM, Components, JSX, Unidirectional data flow, Hooks, Context API, State and Props.

67. **Virtual DOM in ReactJS**

- A lightweight copy of the real DOM that allows React to perform efficient updates by comparing the virtual DOM with the real DOM.

68. **Components in React**
    - Independent, reusable UI elements in React that can either be **functional** or **class-based**.
69. **Class vs Functional Components**
    - **Class Components**: Have lifecycle methods, more complex syntax.
    - **Functional Components**: Simpler, introduced with hooks.
70. **Stateless vs Stateful Components**
    - **Stateless Components**: Don't manage state (just presentational).
    - **Stateful Components**: Manage and store state.
71. **Pure Components**
    - Components that only re-render when their props or state change.
72. **Higher Order Components (HOCs)**
    - A function that takes a component and returns a new component with additional props or logic.
73. **Life Cycle Methods of Class Components**
    - componentDidMount(), componentDidUpdate(), componentWillUnmount(), etc.
74. **Hooks in React**
    - useState, useEffect, useContext, useReducer, useRef, useCallback, useMemo.
75. **Controlled vs Uncontrolled Components**
    - **Controlled Components**: Form elements whose value is controlled by React state.
    - **Uncontrolled Components**: Form elements whose value is managed by the DOM.
76. **setState() Method**
    - Used to update the state of a React component. It triggers a re-render of the component.
77. **State vs Props**
    - **State**: Local to a component and can change.
    - **Props**: Passed from parent to child components and are immutable.
78. **Mutability and How to Achieve It**
    - Mutability refers to the ability to change an object's state after it is created. You can achieve it by modifying properties directly.
79. **Immutability and How to Achieve It**
    - Immutability means the object cannot be changed once created. Achieved by using methods like Object.freeze() or creating copies of arrays/objects.
80. **Data Transfer from Parent to Child Components**
    - Done via props.

81. **Data Transfer from Child to Parent Components**
    - Done using callback functions passed down as props.
82. **Events in ReactJS**
    - Handling events like onClick, onChange, etc., using React's synthetic event system.
83. **Single Page Applications (SPA)**
    - A type of application that loads a single HTML page and dynamically updates it as the user interacts with the app.
84. **Routing in SPA**
    - Managed by libraries like react-router to handle navigation without reloading the page.
85. **Making API Calls**
    - Using fetch, axios, or other libraries.
86. **fetch vs axios**
    - **fetch**: Native JavaScript, returns a promise.
    - **axios**: A promise-based HTTP client that supports interceptors and more features.
87. **Props Drilling**
    - Passing data from a parent to child components through multiple layers of components.
88. **Context API vs Redux**
    - **Context API**: A simpler solution for managing state in React applications.
    - **Redux**: A more complex, scalable solution for state management, especially in large applications.
89. **Thunk vs Saga**
    - **Thunk**: Middleware that allows action creators to return a function instead of an action.
    - **Saga**: A more complex middleware that uses generator functions for handling side effects.
90. **API Requests (GET, POST, PUT, DELETE)**
    - Examples for making API requests using fetch or axios.
91. **Lazy Loading**
    - Loading components or assets only when they are needed.
92. **Code Splitting vs Eager Loading**
    - **Code Splitting**: Loading parts of code on demand.
    - **Eager Loading**: Loading all resources upfront.
93. **Spread vs Rest Operator**
    - **Spread**: Used to expand an iterable (e.g., array) into individual elements.

- **Rest**: Used to collect multiple elements into an array.

94. **Deep Copy vs Shallow Copy**

- **Shallow Copy**: Copies only the top level of an object.
- **Deep Copy**: Recursively copies all levels of an object.

95. **Mutability and Immutability in JS**

- **Mutability**: Objects can be modified.
- **Immutability**: Objects cannot be modified once created.

96. **querySelector() vs querySelectorAll()**

- **querySelector()**: Returns the first element that matches the selector.
- **querySelectorAll()**: Returns all elements that match the selector.

97. **innerHTML vs innerText**

- **innerHTML**: Gets or sets HTML content inside an element.
- **innerText**: Gets or sets the text content inside an element.

98. **Adding Styles Dynamically in JS**

- Modify the style property of an element.
- element.style.color = 'blue';

99. **Adding Elements Dynamically in JS**

- Use createElement() and appendChild().
- const newDiv = document.createElement('div');
- document.body.appendChild(newDiv);

100. The inline, block, and inline-block in HTML

- **inline**: Element takes only as much width as its content.
- **block**: Element takes the full width available.
- **inline-block**: Element is inline but behaves like a block.

101. Margin vs Padding

- **Margin**: Space outside an element.
- **Padding**: Space inside an element, between content and border.

102. Box Model in CSS

- The box model consists of content, padding, border, and margin.

103. Pseudo Classes vs Pseudo Elements

- **Pseudo Classes**: Apply styles based on element state (e.g., :hover).
- **Pseudo Elements**: Apply styles to specific parts of an element (e.g., ::before).

104. Centering Elements in CSS

- Use flex, grid, or absolute positioning.

105. Relative vs Absolute Position

- **Relative**: Positioned relative to its normal position.
- **Absolute**: Positioned relative to its nearest positioned ancestor.

CODING QUESTIONS

1. **Reverse a given number:**

```
public class ReverseNumber {
    public static void main(String[] args) {
        int num = 12345;
        int reversed = 0;
        while (num != 0) {
            int digit = num % 10;
            reversed = reversed * 10 + digit;
            num /= 10;
        }
        System.out.println("Reversed number: " + reversed);
    }
}
```

2. **Palindrome number:**

```
public class PalindromeNumber {
    public static void main(String[] args) {
        int num = 121, originalNum = num, reversed = 0;
        while (num != 0) {
            int digit = num % 10;
            reversed = reversed * 10 + digit;
            num /= 10;
        }
        if (originalNum == reversed) {
            System.out.println("Palindrome");
```

```
        } else {
            System.out.println("Not Palindrome");
        }
    }
}
```

3. **Prime number:**

```
public class PrimeNumber {
    public static void main(String[] args) {
        int num = 29;
        boolean isPrime = true;
        for (int i = 2; i <= num / 2; i++) {
            if (num % i == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            System.out.println(num + " is Prime");
        } else {
            System.out.println(num + " is Not Prime");
        }
    }
}
```

4. **Armstrong number:**

```
public class ArmstrongNumber {
    public static void main(String[] args) {
        int num = 153, originalNum = num, sum = 0, digit;
        while (num != 0) {
            digit = num % 10;
            sum += digit * digit * digit;
            num /= 10;
        }
```

```
        if (originalNum == sum) {
            System.out.println("Armstrong Number");
        } else {
            System.out.println("Not an Armstrong Number");
        }
    }
}
```

5. **Strong number:**

```
public class StrongNumber {
    public static void main(String[] args) {
        int num = 145, originalNum = num, sum = 0;
        while (num != 0) {
            int digit = num % 10;
            sum += factorial(digit);
            num /= 10;
        }
        if (sum == originalNum) {
            System.out.println("Strong Number");
        } else {
            System.out.println("Not a Strong Number");
        }
    }

    public static int factorial(int n) {
        int fact = 1;
        for (int i = 1; i <= n; i++) {
            fact *= i;
        }
        return fact;
    }
}
```

6. **Fibonacci Series:**

```
public class Fibonacci {
    public static void main(String[] args) {
        int n = 10, a = 0, b = 1;
        System.out.print("Fibonacci Series: " + a + " " + b);
        for (int i = 2; i < n; i++) {
            int next = a + b;
            System.out.print(" " + next);
            a = b;
            b = next;
        }
    }
}
```

7. **Factorial:**

```
public class Factorial {
    public static void main(String[] args) {
        int num = 5, result = 1;
        for (int i = 1; i <= num; i++) {
            result *= i;
        }
        System.out.println("Factorial: " + result);
    }
}
```

8. **Reverse Array Elements:**

```
public class ReverseArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        for (int i = arr.length - 1; i >= 0; i--) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

9. **Display Duplicates in the array:**

```
public class DuplicateArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 1};
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j]) {
                    System.out.println("Duplicate: " + arr[i]);
                }
            }
        }
    }
}
```

10. **Remove Duplicates in array:**

```
public class RemoveDuplicates {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 1};
        int[] temp = new int[arr.length];
        int j = 0;
        for (int i = 0; i < arr.length; i++) {
            boolean flag = false;
            for (int k = 0; k < j; k++) {
                if (arr[i] == temp[k]) {
                    flag = true;
                    break;
                }
            }
            if (!flag) {
                temp[j++] = arr[i];
            }
        }
        for (int i = 0; i < j; i++) {
            System.out.print(temp[i] + " ");
```

```
        }
    }
}
```

11. **Display Unique elements in array:**

```
public class UniqueArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 1};
        for (int i = 0; i < arr.length; i++) {
            boolean isUnique = true;
            for (int j = 0; j < arr.length; j++) {
                if (i != j && arr[i] == arr[j]) {
                    isUnique = false;
                    break;
                }
            }
            if (isUnique) {
                System.out.print(arr[i] + " ");
            }
        }
    }
}
```

12. **Push even elements at the beginning of the array and odd elements at the end of the array:**

```
public class EvenOddArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6};
        int[] result = new int[arr.length];
        int j = 0, k = arr.length - 1;

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] % 2 == 0) {
                result[j++] = arr[i];
            } else {
```

```
                result[k--] = arr[i];
            }
        }

        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

13. **Push all 1's at the beginning and all 0's at the end of the array:**

```
public class OneZeroArray {
    public static void main(String[] args) {
        int[] arr = {0, 1, 0, 1, 0};
        int[] result = new int[arr.length];
        int j = 0, k = arr.length - 1;

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == 1) {
                result[j++] = arr[i];
            } else if (arr[i] == 0) {
                result[k--] = arr[i];
            }
        }

        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

14. **Find the equilibrium of array:**

```
public class EquilibriumArray {
    public static void main(String[] args) {
```

```
        int[] arr = {1, 3, 5, 2, 2};
        int totalSum = 0;
        for (int num : arr) {
            totalSum += num;
        }

        int leftSum = 0;
        for (int i = 0; i < arr.length; i++) {
            totalSum -= arr[i];
            if (leftSum == totalSum) {
                System.out.println("Equilibrium index: " + i);
                return;
            }
            leftSum += arr[i];
        }
        System.out.println("No equilibrium index found");
    }
}
```

15. **Rotate by k:**

```
public class RotateArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int k = 2;
        int n = arr.length;
        k = k % n; // Handle rotation larger than array length

        int[] result = new int[n];
        for (int i = 0; i < n; i++) {
            result[(i + k) % n] = arr[i];
        }

        for (int num : result) {
```

```
            System.out.print(num + " ");
        }
    }
}
```

16. **Frequency of elements:**

```
public class FrequencyOfElements {
    public static void main(String[] args) {
        int[] arr = {1, 2, 2, 3, 1, 1};
        for (int i = 0; i < arr.length; i++) {
            int count = 1;
            if (arr[i] != -1) {
                for (int j = i + 1; j < arr.length; j++) {
                    if (arr[i] == arr[j]) {
                        count++;
                        arr[j] = -1; // Marking element as counted
                    }
                }
                System.out.println(arr[i] + " appears " + count + " times");
            }
        }
    }
}
```

17. **Sum of adjacent numbers:**

```
public class AdjacentSum {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4};
        for (int i = 0; i < arr.length - 1; i++) {
            System.out.println(arr[i] + " + " + arr[i + 1] + " = " + (arr[i] + arr[i + 1]));
        }
    }
}
```

18. **Target sum:**

```
public class TargetSum {
    public static void main(String[] args) {
        int[] arr = {2, 7, 11, 15};
        int target = 9;
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] + arr[j] == target) {
                    System.out.println("Indices: " + i + " " + j);
                }
            }
        }
    }
}
```

19. **Unique elements in array:**

```
public class UniqueElements {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 1};
        for (int i = 0; i < arr.length; i++) {
            boolean isUnique = true;
            for (int j = 0; j < arr.length; j++) {
                if (i != j && arr[i] == arr[j]) {
                    isUnique = false;
                    break;
                }
            }
            if (isUnique) {
                System.out.print(arr[i] + " ");
            }
        }
    }
}
```

20. Implement Linear and Binary Search

Linear Search:

```
public class LinearSearch {
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // Return the index of the element
            }
        }
        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] arr = {3, 5, 7, 2, 8, 1};
        int target = 7;
        int result = linearSearch(arr, target);
        System.out.println(result == -1 ? "Element not found" : "Element found at index: " + result);
    }
}
```

Binary Search:

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) {
                return mid; // Element found
            }
            if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
```

```
            }
        }
        return -1; // Element not found
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 5, 7, 8};
        int target = 5;
        int result = binarySearch(arr, target);
        System.out.println(result == -1 ? "Element not found" : "Element found at index: " + result);
    }
}
```

21. Find First Max, Second Max, First Min, Second Min

```
public class FindMaxMin {
    public static void findMaxMin(int[] arr) {
        int max = Integer.MIN_VALUE, secondMax = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE, secondMin = Integer.MAX_VALUE;

        for (int num : arr) {
            if (num > max) {
                secondMax = max;
                max = num;
            } else if (num > secondMax && num != max) {
                secondMax = num;
            }

            if (num < min) {
                secondMin = min;
                min = num;
            } else if (num < secondMin && num != min) {
                secondMin = num;
            }
```

```
        }
        System.out.println("First Max: " + max);
        System.out.println("Second Max: " + secondMax);
        System.out.println("First Min: " + min);
        System.out.println("Second Min: " + secondMin);
    }

    public static void main(String[] args) {
        int[] arr = {1, 4, 2, 7, 9, 5};
        findMaxMin(arr);
    }
}
```

22. Sum of Array Elements

```
public class SumArray {
    public static int sumArray(int[] arr) {
        int sum = 0;
        for (int num : arr) {
            sum += num;
        }
        return sum;
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        System.out.println("Sum of Array Elements: " + sumArray(arr));
    }
}
```

23. Count Vowels, Consonants, and Digits

```
public class CountVowelsConsonantsDigits {
    public static void countVowelsConsonantsDigits(String str) {
        int vowels = 0, consonants = 0, digits = 0;
        str = str.toLowerCase();
```

```
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            if (ch >= 'a' && ch <= 'z') {
                if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
                    vowels++;
                } else {
                    consonants++;
                }
            } else if (ch >= '0' && ch <= '9') {
                digits++;
            }
        }
        System.out.println("Vowels: " + vowels);
        System.out.println("Consonants: " + consonants);
        System.out.println("Digits: " + digits);
    }

    public static void main(String[] args) {
        String str = "Hello123";
        countVowelsConsonantsDigits(str);
    }
}
```

24. Convert Multidimensional Array to Single Dimensional Array (without using methods)

```
public class ConvertMultidimensionalToSingle {
    public static int[] convertToSingleDimensional(int[][] arr) {
        int totalElements = 0;
        // Count total elements in the 2D array
        for (int i = 0; i < arr.length; i++) {
            totalElements += arr[i].length;
        }
        // Create a 1D array
        int[] result = new int[totalElements];
```

```
        int index = 0;
        // Fill the 1D array with elements from the 2D array
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                result[index++] = arr[i][j];
            }
        }
        return result;
    }
    public static void main(String[] args) {
        int[][] arr = {{1, 2, 3}, {4, 5}, {6, 7, 8}};
        int[] result = convertToSingleDimensional(arr);
        System.out.print("Converted Array: ");
        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```