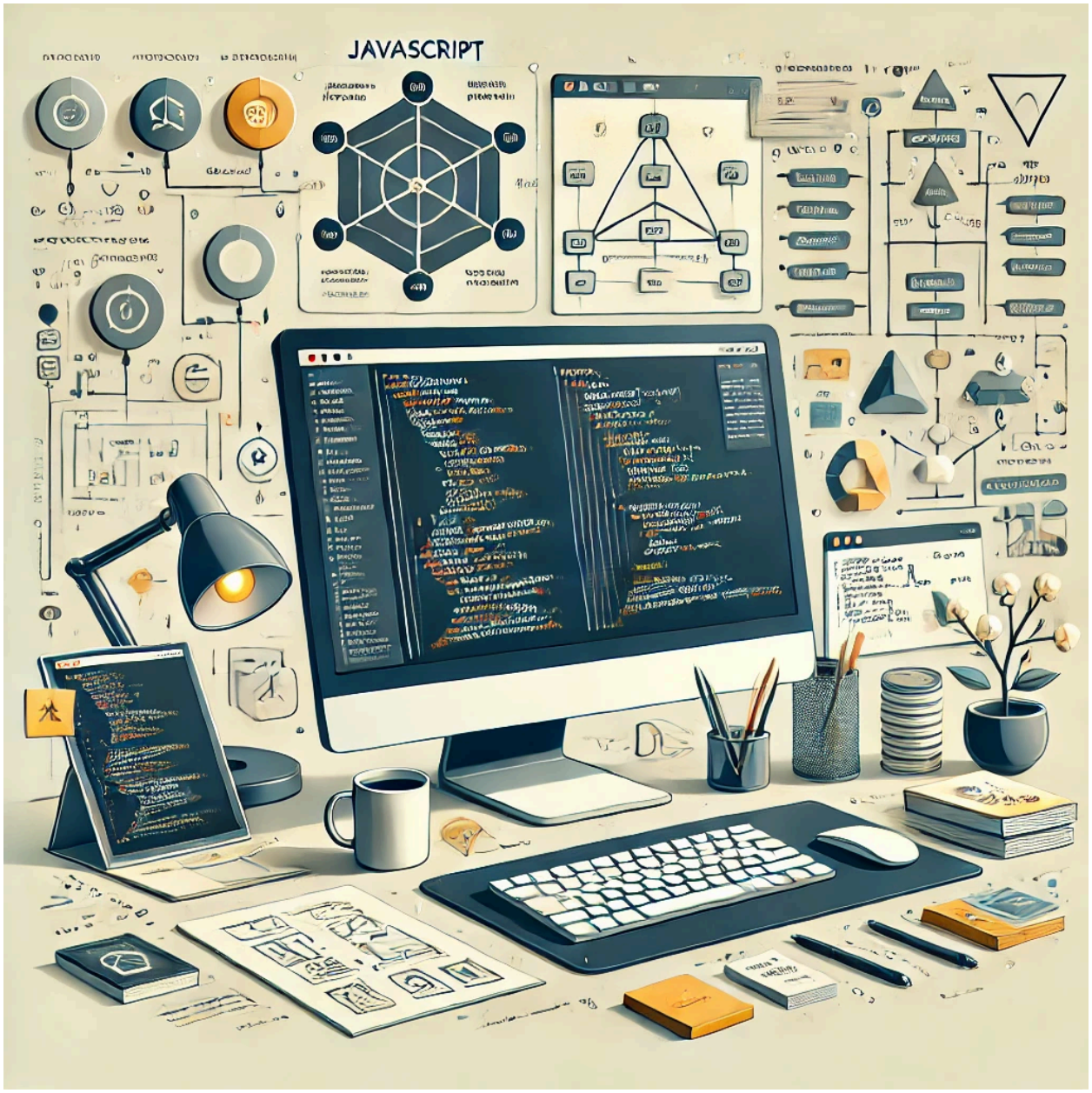
JavaScript Algorithms and Data Structures: Comprehensive Guide



JavaScript Algorithms and Data Structures: Comprehensive Guide 1
What Are Algorithms and Data Structures? 2
Why Learn Algorithms and Data Structures? 2
Common Data Structures 2
1. Arrays 2
2. Linked Lists 3

3. Stacks 4
4. Queues 4
Common Algorithms **5**
1. Searching Algorithms 5
2. Sorting Algorithms 6
Exercises **7**
Exercise 1: Reverse a String Using a Stack 7
Exercise 2: Find the Maximum in a Linked List 7
Exercise 3: Implement a Queue Using Two Stacks 8
Multiple-Choice Questions **8**
Question 1: 8
Question 2: 9
Question 3: 9
Best Practices **9**

This guide introduces essential algorithms and data structures, implemented using JavaScript. You'll learn key concepts, code examples, and practical applications, with exercises and quiz questions to reinforce your understanding.

What Are Algorithms and Data Structures?

- **Algorithms:** Step-by-step procedures or formulas for solving problems.
- **Data Structures:** Organized formats for storing and managing data.

Why Learn Algorithms and Data Structures?

1. Enhance problem-solving skills.
2. Improve application performance.
3. Prepare for coding interviews.

Common Data Structures

1. Arrays

Arrays store multiple values in a single variable.

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[1]); // Output: banana
```

Common Operations:

- Access: `array[index]`
- Insert: `push()`, `unshift()`
- Remove: `pop()`, `shift()`

2. Linked Lists

A linked list is a collection of nodes where each node contains data and a reference to the next node.

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
class LinkedList {
  constructor() {
    this.head = null;
  }
  append(data) {
    const newNode = new Node(data);
    if (!this.head) {
      this.head = newNode;
    } else {
      let current = this.head;
      while (current.next) {
        current = current.next;
      }
      current.next = newNode;
    }
  }
  display() {
    let current = this.head;
    while (current) {
      console.log(current.data);
      current = current.next;
    }
  }
}
```

```
const list = new LinkedList();
list.append(10);
list.append(20);
list.display(); // Output: 10, 20
```

3. Stacks

A stack follows the **LIFO (Last In, First Out)** principle.

```
class Stack {
  constructor() {
    this.items = [];
  }
  push(item) {
    this.items.push(item);
  }
  pop() {
    return this.items.pop();
  }
  peek() {
    return this.items[this.items.length - 1];
  }
}
const stack = new Stack();
stack.push(5);
stack.push(10);
console.log(stack.pop()); // Output: 10
```

4. Queues

A queue follows the **FIFO (First In, First Out)** principle.

```
class Queue {
  constructor() {
    this.items = [];
  }
  enqueue(item) {
    this.items.push(item);
  }
  dequeue() {
```

```
    return this.items.shift();
  }
}
const queue = new Queue();
queue.enqueue(5);
queue.enqueue(10);
console.log(queue.dequeue()); // Output: 5
```

Common Algorithms

1. Searching Algorithms

- **Linear Search:** Iterates through the list to find a value.

```
function linearSearch(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i;
    }
  }
  return -1;
}
console.log(linearSearch([1, 2, 3, 4, 5], 3)); // Output: 2
```

- **Binary Search:** Searches a sorted array by repeatedly dividing the search interval in half.

```
function binarySearch(arr, target) {
  let left = 0;
  let right = arr.length - 1;
  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) {
      return mid;
    } else if (arr[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
```

```
  }
  return -1;
}
console.log(binarySearch([1, 2, 3, 4, 5], 3)); // Output: 2
```

2. Sorting Algorithms

- **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order.

```
function bubbleSort(arr) {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
      }
    }
  }
  return arr;
}
console.log(bubbleSort([5, 3, 8, 4, 2])); // Output: [2, 3, 4, 5, 8]
```

- **Merge Sort:** Divides the array into halves, sorts them, and merges the results.

```
function mergeSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }
  const mid = Math.floor(arr.length / 2);
  const left = mergeSort(arr.slice(0, mid));
  const right = mergeSort(arr.slice(mid));
  return merge(left, right);
}
function merge(left, right) {
  const result = [];
  while (left.length && right.length) {
    if (left[0] < right[0]) {
      result.push(left.shift());
    } else {
      result.push(right.shift());
```

```
    }
  }
  return result.concat(left, right);
}
console.log(mergeSort([5, 3, 8, 4, 2])); // Output: [2, 3, 4, 5, 8]
```

Exercises

Exercise 1: Reverse a String Using a Stack

Write a function to reverse a string using a stack data structure.

Solution:

```
function reverseString(str) {
  const stack = [];
  for (const char of str) {
    stack.push(char);
  }
  return stack.reverse().join('');
}
console.log(reverseString("hello")); // Output: "olleh"
```

Exercise 2: Find the Maximum in a Linked List

Write a function to find the maximum value in a linked list.

Solution:

```
function findMax(linkedList) {
  let max = -Infinity;
  let current = linkedList.head;
  while (current) {
    if (current.data > max) {
      max = current.data;
    }
    current = current.next;
  }
  return max;
}
```

Exercise 3: Implement a Queue Using Two Stacks

Write a function to implement a queue using two stacks.

Solution:

```
class QueueUsingStacks {
  constructor() {
    this.stack1 = [];
    this.stack2 = [];
  }
  enqueue(item) {
    this.stack1.push(item);
  }
  dequeue() {
    if (!this.stack2.length) {
      while (this.stack1.length) {
        this.stack2.push(this.stack1.pop());
      }
    }
    return this.stack2.pop();
  }
}
const queue = new QueueUsingStacks();
queue.enqueue(1);
queue.enqueue(2);
console.log(queue.dequeue()); // Output: 1
```

Multiple-Choice Questions

Question 1:

Which data structure follows the **LIFO** principle?

1. Array
2. Stack
3. Queue
4. Linked List

Answer: 2. Stack

Question 2:

What is the time complexity of binary search?

1. O(1)O(1)O(1)
2. O(n)O(n)O(n)
3. O(logn)O(\log n)O(logn)
4. O(n2)O(n^2)O(n2)

Answer: 3. O(logn)O(\log n)O(logn)

Question 3:

Which sorting algorithm is the most efficient for large datasets?

1. Bubble Sort
2. Selection Sort
3. Merge Sort
4. Insertion Sort

Answer: 3. Merge Sort

Best Practices

1. **Analyze Complexity:** Understand the time and space complexity of your algorithms.
2. **Use Built-in Methods:** Leverage JavaScript's built-in methods for simpler operations (e.g., `sort`, `map`).
3. **Practice Frequently:** Solve problems on platforms like LeetCode or HackerRank.