Common Algorithms:

1. **Sorting Algorithms:**
   - **Bubble Sort:** Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
   - **Merge Sort:** Divides the unsorted list into n sub-lists, repeatedly merges sub-lists to produce new sorted sub-lists.
2. **Searching Algorithms:**
   - **Binary Search:** Divides a sorted array into halves and eliminates half of the remaining elements at each step.
   - **Linear Search:** Iterates through each element in a sequence until a match is found.
3. **Recursion:**
   - A technique in which a function calls itself directly or indirectly.
   - Examples include the factorial function and recursive algorithms on trees and graphs.
4. **Dynamic Programming:**
   - A method for solving complex problems by breaking them down into simpler, overlapping subproblems.
   - Memoization and bottom-up approaches are common.

Common Algorithms

1.Sorting Algorithms

1.1 Bubble Sort

=> Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. It is called bubble sort because smaller elements gradually "bubble" their way to the top of the list.

Here's how bubble sort works in JavaScript:

```javascript
function bubbleSort(arr) {
    const n = arr.length;

    for (let i = 0; i < n - 1; i++) {
        for (let j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
            }
        }
    }

    return arr;
}

// Example usage
const array = [5, 3, 8, 4, 2];
console.log( data[0]: "Original array:",  data[1]: array);
console.log( data[0]: "Sorted array:",  data[1]: bubbleSort(array));
```

In this implementation:

- We start with an outer loop that runs from `i = 0` to `n - 1`, where `n` is the length of the array.
- Inside the outer loop, we have an inner loop that runs from `j = 0` to `n - 1 - i`. This is because after each iteration of the outer loop, the largest element will be bubbled to its correct position at the end of the array, so we don't need to compare it again.
- Inside the inner loop, we compare adjacent elements `arr[j]` and `arr[j + 1]`. If `arr[j]` is greater than `arr[j + 1]`, we swap them.
- We repeat this process until the entire array is sorted.

Time Complexity:

- In the worst-case scenario, where the array is in reverse order, bubble sort will perform **O**(**n**2) comparisons and swaps, where **n** is the number of elements in the

array.

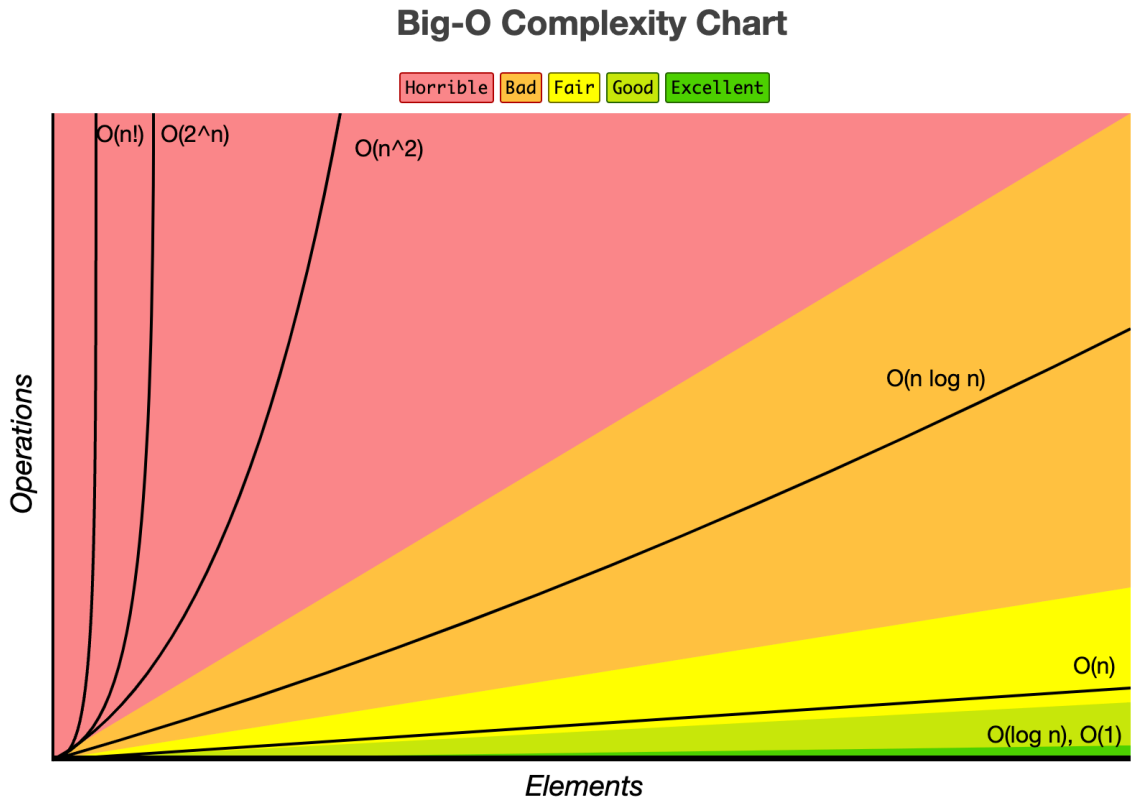
- In the best-case scenario, where the array is already sorted, bubble sort will perform **O**(**n**) comparisons and **O**(**1**) swaps.
- On average, bubble sort performs **O**(**n**2) comparisons and swaps.

Space Complexity:

- Bubble sort has a space complexity of **O**(**1**) because it only requires a constant amount of additional space for storing temporary variables.

Bubble sort is not efficient for large datasets due to its quadratic time complexity. However, it is easy to understand and implement, making it useful for educational purposes or for sorting small datasets where performance is not critical.

***BigO cheetSheet



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

1.2 Merge Sort

=> Merge sort is a classic divide-and-conquer sorting algorithm known for its efficiency and stability. It divides the unsorted array into smaller sub-arrays until each sub-array

contains only one element. Then, it merges those sub-arrays back together, sorting them as it does so.

Here's an explanation of merge sort in JavaScript along with the analysis of its time and space complexity:

```javascript
function mergeSort(arr) {
    if (arr.length <= 1) {
        return arr; // Base case: if the array has 0 or 1 element, it is already sorted
    }

    // Divide the array into two halves
    const mid = Math.floor( x: arr.length / 2);
    const leftHalf = arr.slice(0, mid);
    const rightHalf = arr.slice(mid);

    // Recursively sort each half
    const sortedLeftHalf = mergeSort( arr: leftHalf);
    const sortedRightHalf = mergeSort( arr: rightHalf);

    // Merge the sorted halves
    return merge( left: sortedLeftHalf,  right: sortedRightHalf);
}

// Merge function to merge two sorted arrays
function merge(left, right) {
    let result = [];
    let leftIndex = 0;
    let rightIndex = 0;

    // Compare elements from both arrays and add the smaller one to the result array
    while (leftIndex < left.length && rightIndex < right.length) {
        if (left[leftIndex] < right[rightIndex]) {
            result.push( items[0]: left[leftIndex]);
            leftIndex++;
        } else {
            result.push( items[0]: right[rightIndex]);
            rightIndex++;
        }
    }

    // Add remaining elements from left array
    while (leftIndex < left.length) {
        result.push( items[0]: left[leftIndex]);
        leftIndex++;
    }

    // Add remaining elements from right array
    while (rightIndex < right.length) {
        result.push( items[0]: right[rightIndex]);
        rightIndex++;
    }

    return result;
}

// Example usage
const array = [5, 3, 8, 4, 2];
console.log( data[0]: "Original array:",  data[1]: array);
console.log( data[0]: "Sorted array:",  data[1]: mergeSort(array));
```

Explanation:

- `mergeSort` function: This is the main function that implements the merge sort algorithm. It takes an array `arr` as input. If the array has 0 or 1 element, it is already

sorted, so the function simply returns the array. Otherwise, it divides the array into two halves (`leftHalf` and `rightHalf`), recursively sorts each half, and then merges them using the `merge` function.

- `merge` function: This function takes two sorted arrays `left` and `right` as input and merges them into a single sorted array. It uses two pointers (`leftIndex` and `rightIndex`) to iterate through the elements of the two arrays, comparing them and adding the smaller one to the `result` array. Once one of the arrays is fully processed, any remaining elements in the other array are added to the `result`.

Time Complexity:

- Merge sort has a time complexity **O**(**n**log**n**) in all cases, where **n** is the number of elements in the array. This is because the array is repeatedly divided into halves until each sub-array contains only one element, which takes **O**(**log**n) time, and then the merging process takes **O**(**n**) time.

Space Complexity:

- Merge sort has a space complexity of **O**(**n**) because it requires additional space to store the two halves of the array during the recursion. This additional space is proportional to the size of the input array.

2.Searching Algorithms

2.1 Binary Search

=> Binary search is an efficient algorithm for finding a target value within a sorted array. It works by repeatedly dividing the search interval in half and narrowing down the possible locations of the target value until it is found.

Here's an explanation of binary search in JavaScript along with the analysis of its time and space complexity:

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

    while (left <= right) {
        // Find the middle index of the array
        let mid = Math.floor( x: (left + right) / 2);

        // If the target is found at the middle, return the index
        if (arr[mid] === target) {
            return mid;
        }

        // If the target is less than the middle element, search the left half
        if (target < arr[mid]) {
            right = mid - 1;
        }
        // If the target is greater than the middle element, search the right half
        else {
            left = mid + 1;
        }
    }

    // If the target is not found, return -1
    return -1;
}

// Example usage
const array = [1, 2, 3, 4, 5, 6, 7, 8, 9];
const target = 5;
console.log( data[0]: "Index of target element:",  data[1]: binarySearch(array, target));
```

Explanation:

- `binarySearch` function: This is the main function that implements the binary search algorithm. It takes a sorted array `arr` and a target value `target` as input. It initializes two pointers `left` and `right` to the beginning and end of the array, respectively. It then enters a loop where it calculates the middle index (`mid`) of the current search interval and checks if the target value is equal to the value at the middle index. If the target value is found, the function returns the index of the target element. Otherwise, it updates the search interval by moving either the `left` or `right` pointer based on whether the target value is less than or greater than the value at the middle index. The loop continues until the search interval is empty (`left` is greater than `right`), indicating that the target value is not present in the array.

Time Complexity:

- Binary search has a time complexity of **O**(**log**n), where **n** is the number of elements in the array. This is because at each step, the search interval is divided in half, leading to a significant reduction in the number of elements to be searched.

Space Complexity:

- Binary search has a space complexity of **O**(**1**) because it only requires a constant amount of additional space for storing temporary variables. It does not require any additional data structures that grow with the size of the input array.

2.1 Linear Search

=> Linear search is a simple search algorithm that sequentially checks each element in a collection until a match is found or the whole collection has been searched. It is also known as a sequential search.

Here's an explanation of linear search in JavaScript along with the analysis of its time and space complexity:

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

// Example usage
const array = [5, 3, 8, 4, 2];
const target = 8;
console.log( data[0]: "Index of target element:",  data[1]: linearSearch(array, target));
```

Explanation:

- `linearSearch` function: This is the main function that implements the linear search algorithm. It takes an array `arr` and a target value `target` as input. It iterates through each element of the array using a `for` loop. At each iteration, it compares the current element with the target value. If a match is found, it returns the index of the element. If the entire array is traversed and no match is found, it returns -1 to indicate that the target value is not present in the array.

Time Complexity:

- In the worst-case scenario, where the target value is not present in the array or is present at the last position, linear search has a time complexity of **O**(**n**), where **n** is the number of elements in the array. This is because it may need to iterate through the entire array to find the target value.

- In the best-case scenario, where the target value is found at the first position, linear search has a time complexity of **O(1)**. However, this is an unlikely scenario and is not typical.

Space Complexity:

- Linear search has a space complexity of **O(1)** because it only requires a constant amount of additional space for storing temporary variables. It does not require any additional data structures that grow with the size of the input array.

3. Recursion

=> Recursion is a programming technique where a function calls itself in order to solve a problem. It's a fundamental concept in computer science and is widely used in various algorithms and data structures. Recursive functions break down a problem into smaller subproblems and solve each subproblem recursively until a base case is reached, at which point the function stops calling itself and returns a result.

Here's an explanation of recursion in JavaScript along with the analysis of its time and space complexity:

```
function factorial(n) {
    // Base case: if n is 0 or 1, return 1
    if (n === 0 || n === 1) {
        return 1;
    } else {
        // Recursive case: n * factorial(n - 1)
        return n * factorial( n: n - 1);
    }
}

// Example usage
console.log( data[0]: factorial(5)); // Output: 120 (5! = 5 * 4 * 3 * 2 * 1 = 120)
```

Explanation:

- `factorial` function: This is a recursive function that calculates the factorial of a non-negative integer `n`. The factorial of a non-negative integer `n`, denoted as `n!`, is the product of all positive integers less than or equal to `n`. The function has a base case where if `n` is 0 or 1, it returns 1. Otherwise, it recursively calls itself with `n - 1` and multiplies the result by `n`.

Time Complexity:

- The time complexity of a recursive function depends on the number of recursive calls made and the work done at each call. In the case of the `factorial` function, each recursive call reduces the value of `n` by 1 until the base case is reached. Therefore, the time complexity of the `factorial` function is **O**(**n**), where **n** is the value of the input parameter.

Space Complexity:

- The space complexity of a recursive function depends on the number of recursive calls made and the space required to store the call stack. In the case of the `factorial` function, since there are **n** recursive calls made (one for each integer from **n** down to 1), the space complexity is also **O**(**n**) due to the size of the call stack. However, in many cases, tail recursion optimization can be applied to reduce the space complexity to **O**(**1**) by reusing the same stack frame for each recursive call. Unfortunately, JavaScript does not currently optimize tail recursion.

4. Dynamic Programing

=> Dynamic programming (DP) is a powerful problem-solving technique used to solve problems by breaking them down into simpler subproblems and solving each subproblem only once. It's especially useful for optimization problems where the solution can be built incrementally.

Here's an explanation of dynamic programming in JavaScript along with the analysis of its time and space complexity:

Dynamic programming typically involves two approaches: memoization and tabulation.

1. **Memoization** :

- In memoization, we store the results of expensive function calls and return the cached result when the same inputs occur again.
- We can use memoization to optimize recursive algorithms by storing the results of subproblems in a data structure (such as an object or an array) and reusing those results instead of recomputing them.
- This approach is particularly useful when the recursive algorithm has overlapping subproblems, meaning the same subproblems are solved repeatedly.

1. **Tabulation** :

- In tabulation, we solve the problem bottom-up by iteratively filling a table or array.
- We start with the simplest subproblems and gradually build up to the desired solution.
- This approach is often used when the problem can be naturally expressed as a series of overlapping subproblems.

Here's an example of dynamic programming in JavaScript using the Fibonacci sequence:

```
// Fibonacci function using memoization
const memo = {};
function fibonacci(n) {
    if (n in memo) {
        return memo[n];
    }
    if (n <= 1) {
        return n;
    }
    memo[n] = fibonacci( n: n - 1) + fibonacci( n: n - 2);
    return memo[n];
}

// Example usage
console.log( data[0]: fibonacci(6)); // Output: 8
```

Time Complexity :

- With memoization or tabulation, the time complexity of dynamic programming algorithms depends on the number of distinct subproblems solved.
- In the case of the Fibonacci sequence using memoization, the time complexity is **O**(**n**), where **n** is the input parameter.
- Tabulation-based dynamic programming algorithms also typically have a time complexity of **O**(**n**) because they involve filling up a table of size **n** in a bottom-up manner.

Space Complexity :

- The space complexity of dynamic programming algorithms depends on whether memoization or tabulation is used and the size of the data structure used to store the results of subproblems.
- In the case of memoization, the space complexity is **O**(**n**) due to the space required to store the results of subproblems in the memoization table.

- In the case of tabulation, the space complexity is also **O**(**n**) due to the space required to store the table or array used to solve subproblems iteratively.