Space and Time Complexity

What are Data Structures and Algorithms?

- **Data Structure**: A way of organizing and storing data so that it can be accessed and modified efficiently. Data structures define the layout of data in memory and help improve the performance of operations like searching, sorting, and inserting.
  - **Example**:
    - **Array**: Stores elements in contiguous memory locations.

      ```
      int[] numbers = {1, 2, 3, 4, 5};
      ```

    - **Linked List**: A list of elements where each element (node) contains the data and a pointer/reference to the next node.

      ```
      class Node
      {
          int data;
          Node next;
          Node(int data) {
          this.data = data;
          this.next = null;
      ```

```
    }
}
```

- **Algorithm**: A finite sequence of well-defined instructions to solve a problem or perform a task. Algorithms are focused on optimizing time and space.
  - **Example**:
    - **Searching**: Linear search (checks each element one by one) vs Binary search (divides and conquers on a sorted array).

```
// Linear search example
public int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) return i;
    }
    return -1;
}

// Binary search example
public 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;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

- **Why study DSA?**:

- **Optimization**: DSA helps design efficient algorithms that use fewer resources (time and space).
- **Coding Interviews**: Most technical interviews for software engineering positions focus heavily on DSA for problem-solving.
- **Real-world Applications**: Understanding DSA enables better system design, from databases to networks and more.

Asymptotic Notation

Asymptotic Notation is a way to describe the efficiency of an algorithm in simple terms, especially when the input size becomes very large. It helps us understand how the running time or memory usage of an algorithm grows as the input size increases, without worrying about the exact details of the code or hardware.

Key Types of Asymptotic Notations:

1. **Big O (O):** Describes the *worst-case* scenario or the upper bound of how the algorithm performs as the input size increases.
2. **Omega (Ω):** Describes the *best-case* scenario or the lower bound.
3. **Theta (Θ):** Describes the *average-case* or how the algorithm performs generally as input grows.

Example:

Let's say you have two algorithms that sort a list of numbers.

1. **Algorithm 1:** Takes about `n` steps to sort the list (linear time complexity).
2. **Algorithm 2:** Takes about `n²` steps to sort the same list (quadratic time complexity).

Now, if the input size (n) is small, both algorithms may perform similarly. But if the input size is very large, Algorithm 1 (with time complexity O(n)) will perform much better because its number of steps grows slower than Algorithm 2 ($O(n^2)$).

Example Code:

- **O(n):**

```
for (int i = 0; i < n; i++) {
    // Does something once per element
}
```

Here, the loop runs `n` times, so the time complexity is **O(n)**.

- **O(n²):**

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // Does something for each pair of elements
    }
}
```

Here, two loops run `n * n` times, so the time complexity is **O(n²)**.

In summary, Asymptotic Notation helps us compare the efficiency of algorithms as input grows large, giving us a clear idea of how scalable they are.

Time and Space Complexity

- **Time Complexity**: Measures the amount of time an algorithm takes to run as a function of input size.
- **Space Complexity**: Measures the amount of memory an algorithm uses during execution.
- **Big O Notation**: Represents the worst-case scenario of an algorithm's time or space complexity, abstracting away constants and lower-order terms.

Common Complexities

1. O(1) — Constant Time Complexity

- **Definition:** The algorithm takes the same amount of time regardless of the input size.
- **Example:** Accessing an element in an array by its index.

Code Example (Array Access):

```
int[] arr = {1, 2, 3, 4, 5};
int element = arr[2];  // Accessing the 3rd element
```

- **Explanation:** No matter how large the array is, accessing any element by its index will always take the same constant time. Therefore, the time complexity is **O(1)**.

2. O(n) — Linear Time Complexity

- **Definition:** The time grows linearly with the size of the input.
- **Example:** A `for` loop that iterates through all elements in a list.

Code Example (Linear Search):

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

- **Explanation:** The algorithm goes through each element one by one, so the number of steps grows directly in proportion to the input size. If there are 10 elements, it checks 10 times. If there are 1000 elements, it checks 1000 times, hence **O(n)**.

3. O(log n) — Logarithmic Time Complexity

- **Definition:** The algorithm reduces the problem size in half at each step.
- **Example:** Binary search, which works on sorted arrays.

Code Example (Binary Search):

```
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;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

- **Explanation:** Binary search divides the input size in half after each comparison. Therefore, if the input size doubles, the number of comparisons only increases by one. For example, in an array of size 1000, binary search only needs about 10 comparisons, which is **logarithmic** growth. Hence, the complexity is **O(log n)**.

4. O(n log n) — Log-linear Time Complexity

- **Definition:** These algorithms have a linear number of operations, each of which takes logarithmic time.
- **Example:** Merge Sort and Quick Sort (on average) are good examples.

Code Example (Merge Sort):

```
void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);  // Merge the two halve
s
    }
}
```

- **Explanation:** Merge Sort splits the array in half recursively and then merges the sorted halves. The splitting takes **log n** time (because the array is divided in half repeatedly), and merging takes **n** time at each level. This results in **O(n log n)** time complexity, which is much faster than **O(n^2)** for larger inputs.

5. O(n^2) — Quadratic Time Complexity

- **Definition:** The time grows quadratically with the size of the input, typically seen with algorithms that involve nested loops.
- **Example:** Bubble Sort or Selection Sort.

Code Example (Bubble Sort):

```
void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

- **Explanation:** In Bubble Sort, the outer loop runs `n` times and the inner loop also runs `n` times for each pass. This results in **n * n = n²** steps, so the time complexity is **O(n²)**. For a small number of elements, this may not be a problem, but as the input grows, the time taken increases significantly.

6. O(2^n) — Exponential Time Complexity

- **Definition:** The time grows exponentially with the size of the input, meaning it doubles with each additional input.
- **Example:** Recursive algorithms that solve a problem by breaking it into multiple smaller subproblems, such as calculating Fibonacci numbers using recursion.

Code Example (Naive Recursive Fibonacci):

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- **Explanation:** In this recursive Fibonacci function, each call to `fibonacci(n)` results in two further calls to `fibonacci(n-1)` and `fibonacci(n-2)`. As `n` increases, the number of recursive calls grows exponentially, resulting in **O(2^n)** time complexity. This becomes impractical for even moderate values of `n`.

7. O(n!) — Factorial Time Complexity

- **Definition:** The time grows factorially with the size of the input, often seen in algorithms that involve generating all possible permutations of a set.
- **Example:** The brute-force solution to the Traveling Salesman Problem (TSP), where we try all possible routes between cities.

Code Example (Permutations):

```
void permute(char[] arr, int l, int r) {
    if (l == r) {
        System.out.println(Arrays.toString(arr));
    } else {
        for (int i = l; i <= r; i++) {
            swap(arr, l, i);
            permute(arr, l+1, r);
            swap(arr, l, i);  // Backtrack
        }
    }
}
```

- **Explanation:** In this algorithm, each element has `n!` possible permutations, meaning the number of possible solutions grows factorially as the input size increases. For an input of size `n`, the algorithm has **O(n!)** time complexity, which is extremely inefficient for large inputs.

Summary Table of Common Time Complexities:

Time Complexity	Example Algorithm	Description
O(1)	Accessing an element in an	Constant time, no matter the input size

	array	
O(n)	Linear Search, Single loops	Grows linearly with the input size
O(log n)	Binary Search	Halves the problem size with each step
O(n log n)	Merge Sort, Quick Sort	Log-linear time, used in efficient sorting
O(n²)	Bubble Sort, Selection Sort	Quadratic time, nested loops
O(2^n)	Recursive Fibonacci	Exponential time, doubles with each added input
O(n!)	Permutations, TSP	Factorial time, grows very quickly with input size