ChatGPT

Time and Space Complexity

Time Complexity

- Measures how the **runtime of an algorithm** increases with input size `n`.
- Expressed as functions like `O(1)`, `O(n)`, `O(log n)`, etc.

Space Complexity

- Measures how much **extra memory** (not input) an algorithm uses as input size grows.

Asymptotic Notation: Big O, Big Theta, Small o, etc.

Notation	Meaning	Description
Big O (``)	**Upper Bound**	Worst-case time: Algorithm **won't take longer than** this.
Big Theta (``)	**Tight Bound**	Average/Typical case: Algorithm **always takes about** this much time.
Big Omega (``)	**Lower Bound**	Best-case time: Algorithm **takes at least** this much time.
Small o (``)	**Strict Upper Bound**	Grows **faster than algorithm**, but algorithm is never equal to it.
Small omega (``)	**Strict Lower Bound**	Grows **slower than algorithm**, but never equal.
Big Sigma (``)	**Summation**	Used in math, e.g., `Σ(i=1 to n) i = n(n+1)/2`, not time complexity.
Small sigma (``)	No standard meaning in algorithm analysis.	

Time Complexity Graph (Big O)

Here's how **common time complexities** grow as input `n` increases:

```
Time ↑
|
|               O(2ⁿ)      ← Exponential
|             /
|         O(n²)            ← Quadratic
|       /
|     O(n log n)           ← Log-linear
|   /
|  O(n)                    ← Linear
| /
| O(log n)                 ← Logarithmic
|/
|O(1)                      ← Constant
├────────────────────────────→ Input Size (n)
```

Examples & How Time Complexity is Calculated

1. Constant Time – O(1)

```
int getFirst(int[] arr) {
    return arr[0];
}
```

- Always 1 step → O(1)

---

2. Linear Time – O(n)

```
int sum(int[] arr) {
    int total = 0;
    for (int i = 0; i < arr.length; i++) {
        total += arr[i];
    }
    return total;
}
```

- Loop runs `n` times → O(n)

---

3. Quadratic Time – O(n²)

```
void printPairs(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
            System.out.println(arr[i] + ", " + arr[j]);
        }
    }
}
```

- Two nested loops → n × n → O(n²)

---

4. Logarithmic Time – O(log n)

```
int binarySearch(int[] arr, int key) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid;
        else if (arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

- Each step halves the array → O(log n)

---

5. Linearithmic Time – O(n log n)

- Merge Sort:

```
void mergeSort(int[] arr, int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}
```

- Divides into halves → log n
- Merges each part → n

- Total: `n log n` → O(n log n)

---

6. Exponential Time – $O(2^n)$

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

- Each call spawns 2 more → exponential growth → $O(2^n)$

Space Complexity Examples

1. O(1) Space

```
int sum(int[] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];  // just one variable
    }
    return sum;
}
```

2. O(n) Space

```
int[] copy(int[] arr) {
    int[] copy = new int[arr.length];
    for (int i = 0; i < arr.length; i++) {
        copy[i] = arr[i];
    }
    return copy;
}
```
