**Time and space complexity**

**Time complexity –**

It is not the actual time taken but amount of time taken as function of input size(n).

Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the input size (**n**).

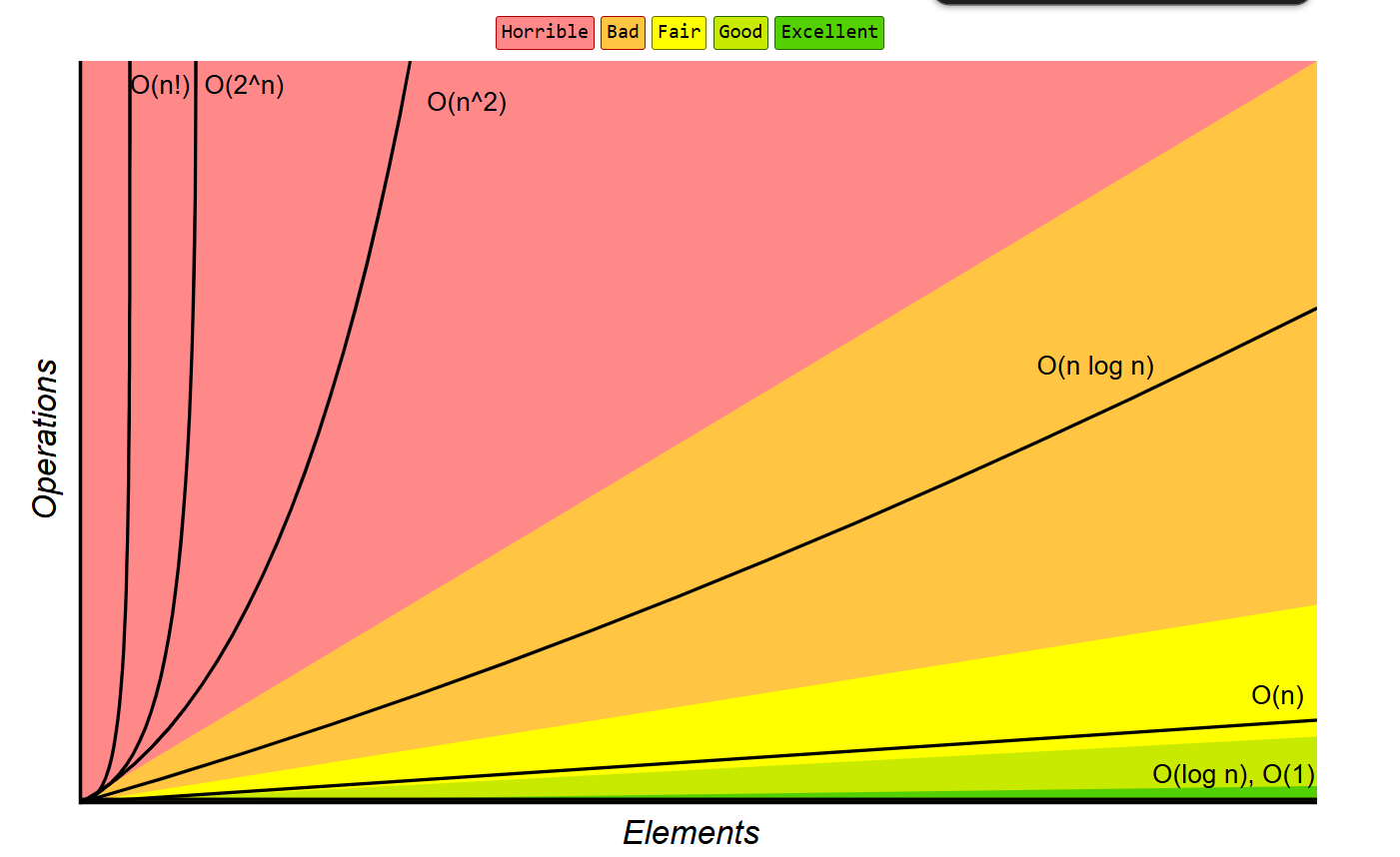
It describes the growth rate of an algorithm's execution time when the input size increases.

It is commonly expressed using **Big O notation (O)**, which represents the worst-case scenario.

**Definition –** “Time complexity is the rate at which the number of operations performed by an algorithm grows relative to the input size n”

**Common Time Complexities:**

* **O(1) - Constant Time:** Execution time remains the same regardless of input size.
* **O(log n) - Logarithmic Time:** Execution time grows logarithmically as input size increases.
* **O(n) - Linear Time:** Execution time grows proportionally to input size.
* **O(n²) - Quadratic Time:** Execution time grows proportionally to the square of input size.
* **O(2ⁿ) - Exponential Time:** Execution time doubles with each additional input element.



**Big O Notation –**

Big O notation is a mathematical notation used to describe the upper bound (worst-case scenario) of an algorithm's time complexity.

It helps in analysing how the execution time or space requirements of an algorithm grow as the input size n increases.

**Why is Big O Notation Important?**

* It provides a standard way to compare different algorithms.
* Helps in understanding the efficiency of an algorithm.
* Focuses on the most significant term and ignores constants and lower-order terms.

**Common Time Complexities:**

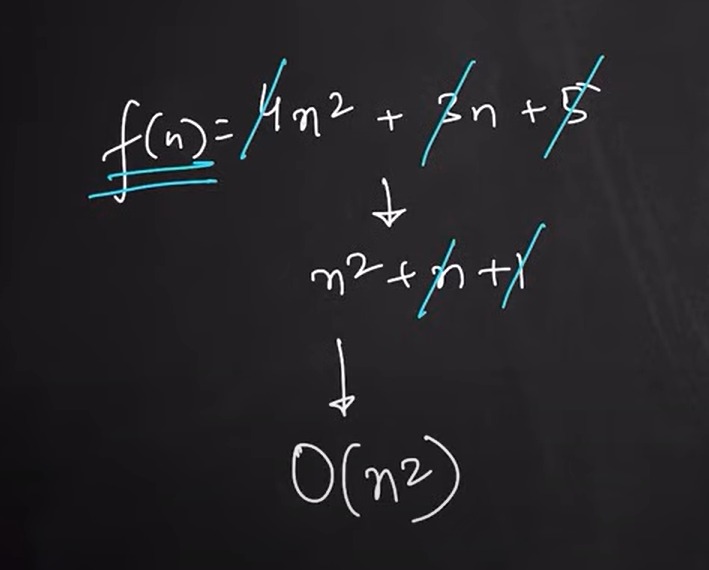
* **O(1) - Constant Time:** Execution time remains the same regardless of input size.
* **O(log n) - Logarithmic Time:** Execution time grows logarithmically as input size increases.
* **O(n) - Linear Time:** Execution time grows proportionally to input size.
* **O(n²) - Quadratic Time:** Execution time grows proportionally to the square of input size.
* **O(2ⁿ) - Exponential Time:** Execution time doubles with each additional input element.

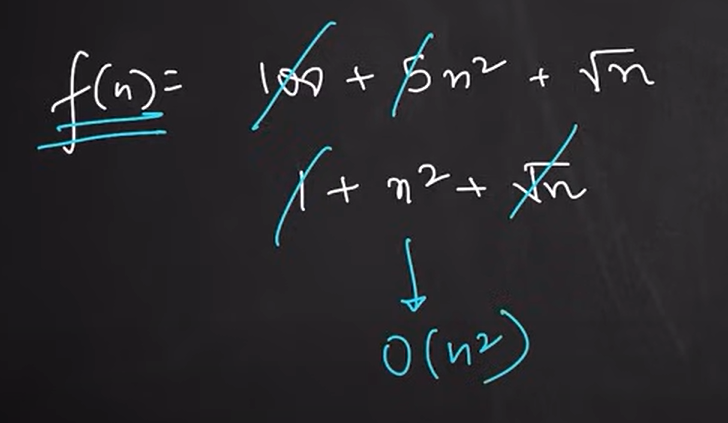
How to calculate Time complexity –

Step 1 – ignore constants (any thing that is a mathematical no. is constant).

Step 2 – only consider the largest term

Example – f(n) = 4n^2 + 2n + 5





**Theta (Θ) and Omega (Ω) Notation in Time Complexity**

While **Big O (O)** notation represents the **worst-case** time complexity, **Theta (Θ)** and **Omega (Ω)** provide a more complete analysis of an algorithm’s performance in different scenarios.

**1. Big O (O) - Worst Case**

* Represents the **upper bound** of an algorithm.
* Ensures the algorithm **won’t take longer** than a certain amount of time.
* Used to analyze the **worst-case scenario** of an algorithm.
* **Example:** Sorting an already sorted array using Bubble Sort is **O(n²)** in the worst case.

**Think of Big O as the "maximum time an algorithm can take."**

**2. Omega (Ω) - Best Case**

* Represents the **lower bound** of an algorithm.
* Ensures the algorithm **will take at least** a certain amount of time.
* Used to analyze the **best-case scenario** of an algorithm.
* **Example:** Insertion Sort has **Ω(n)** in the best case when the array is already sorted.

**Think of Omega as the "minimum time an algorithm will take."**

**3. Theta (Θ) - Average Case / Tight Bound**

* Represents both **upper and lower bounds** of an algorithm.
* Ensures that the algorithm will always run within a specific range of time.
* Used to analyze the **average-case scenario** when input distribution is unknown.
* **Example:** Merge Sort always runs in **Θ(n log n)** time, no matter the input.

**Think of Theta as "the actual runtime when the best and worst case match."**

**Comparison Table**

|  |  |  |
| --- | --- | --- |
| **Notation** | **Definition** | **Scenario** |
| **O (Big O)** | Upper bound (Worst-case) | "At most this much time" |
| **Ω (Omega)** | Lower bound (Best-case) | "At least this much time" |
| **Θ (Theta)** | Tight bound (Average-case) | "Always this much time" |

**Space Complexity –**

Amount of space taken by an algorithm as function of input size(n)

**Space Complexity** refers to the amount of memory an algorithm requires to run as a function of the input size nnn. It includes both:

1. **Fixed Part**: Memory required for constants, variables, and program instructions.
2. **Variable Part**: Memory required for dynamically allocated memory like arrays, recursion stack, etc.

**Formula:**

Total Space Complexity=Fixed Space+Variable Space

**Why is Space Complexity Important?**

* Helps optimize memory usage.
* Essential for memory-limited environments.
* Prevents unnecessary crashes due to memory overflow.

**Common Space Complexities in Big O Notation**

|  |  |  |
| --- | --- | --- |
| **Space Complexity** | **Description** | **Example** |
| **O(1) - Constant** | Uses a fixed amount of memory, regardless of input size. | Swapping two numbers. |
| **O(n) - Linear** | Memory usage grows proportionally with input size. | Storing an array of size nnn. |
| **O(n²) - Quadratic** | Memory increases quadratically. | Using a 2D matrix. |
| **O(log n) - Logarithmic** | Memory increases logarithmically. | Recursive binary search. |
| **O(n!) - Factorial** | Extremely high memory usage. | Storing all permutations of a set. |

**Example in Java**

**1. O(1) - Constant Space**

java

public class ConstantSpace {

public static void main(String[] args) {

int a = 5, b = 10, sum; // Only 3 variables used → O(1)

sum = a + b;

System.out.println(sum);

}

}

🔹 **Explanation**: Memory usage does not depend on input size.

**2. O(n) - Linear Space (Using Arrays)**

java

public class LinearSpace {

public static void main(String[] args) {

int n = 5;

int[] arr = new int[n]; // Memory depends on input size → O(n)

for (int i = 0; i < n; i++) {

arr[i] = i;

}

System.out.println(arr[0]);

}

}

🔹 **Explanation**: If n=1000n = 1000n=1000, memory usage increases proportionally.

**3. O(n) - Linear Space (Recursion Example)**

java

public class RecursiveSpace {

static void recursiveFunction(int n) {

if (n == 0) return;

System.out.println(n);

recursiveFunction(n - 1); // Each recursive call takes extra memory → O(n)

}

public static void main(String[] args) {

recursiveFunction(5);

}

}

🔹 **Explanation**: Each function call adds a new stack frame, requiring **O(n)** space.

**4. O(log n) - Logarithmic Space (Binary Search Example)**

java

public class LogarithmicSpace {

static int binarySearch(int[] arr, int low, int high, int key) {

if (low > high) return -1;

int mid = (low + high) / 2;

if (arr[mid] == key) return mid;

if (arr[mid] > key)

return binarySearch(arr, low, mid - 1, key); // O(log n) recursive calls

return binarySearch(arr, mid + 1, high, key);

}

public static void main(String[] args) {

int[] arr = {1, 3, 5, 7, 9};

System.out.println(binarySearch(arr, 0, arr.length - 1, 7));

}

}

🔹 **Explanation**: Each recursive call halves the problem size → **O(log n) space**.

**Key Takeaways**

* **Iterative algorithms** generally use **O(1) space**.
* **Recursive algorithms** may use **O(n) space** due to stack memory.
* **Sorting algorithms** like Merge Sort use **O(n) space**, while Quick Sort uses **O(log n) space** on average.