## 🧠 Time Complexity Simplified

## ❓ What is Time Complexity?

Let’s take a simple real-life example to demystify time complexity. Suppose you have 5 friends and you want to find out who runs the fastest. You ask each to run and record their times—the one with the lowest time is the winner.  
  
Similarly, time complexity helps us measure how fast or slow a program performs a task. It gives an estimate of the execution time based on input size. Understanding it allows us to analyze and improve code efficiency.

## 🧾 Notations of Time Complexity:

There are 3 notations to describe time complexity:  
  
1. Ω (Omega) – Best-case: e.g., already sorted array.  
2. θ (Theta) – Average-case: average over all input cases.  
3. O (Big-O) – Worst-case: e.g., reverse-sorted array.  
  
Note: By default, we use Big-O unless mentioned otherwise, since it tells us the upper bound of time required.

## 📊 Types of Time Complexities:

1. ⚡ Constant – O(1): Execution time stays constant regardless of input.  
2. 📉 Logarithmic – O(logn): Time increases slowly as input grows.  
3. 📈 Linear – O(n): Time grows directly in proportion to input size.  
4. 📉📈 Linear Logarithmic – O(nlogn): Combination of linear and logarithmic growth.  
5. 🔁 Quadratic – O(n²): Time increases as the square of input size.  
6. 🔃 Cubic – O(n³): Time grows with the cube of the input size.  
7. 🚀 Exponential – O(aⁿ): Time explodes rapidly with input size.

## 🪜 Order of Time Complexities:

From fastest to slowest growth:  
  
O(1) < O(logn) < O(n) < O(nlogn) < O(n²) < O(n³) < O(aⁿ)  
  
The higher in the order, the more time-consuming the program is.

## ✅ Conclusion:

Understanding Time Complexity is essential for writing efficient code. It not only helps in identifying bottlenecks but also prepares you for technical interviews and real-world problem solving. Mastering this concept will help you choose the most optimal approach, saving time and computing resources. 🚀