Comparing Algorithmic Efficiency: $O(n)$ vs $O(\log n)$ and the Effect of Lower-Order Terms

1. Comparing Linear and Logarithmic Complexity

Suppose we have two algorithms with input size n:

- Algorithm A: $T_A(n) = n$ total computations.
- Algorithm B: $T_B(n) = \log_2 n$ total computations.

Now imagine that each computation in Algorithm A takes **1 unit of time**, and each computation in Algorithm B takes **100 units of time**. Then the actual time cost is:

$$\text{Time}_A(n) = n \cdot 1 = n, \quad \text{Time}_B(n) = 100 \cdot \log_2 n$$

Observation

Even though each step of Algorithm B is more expensive, for large n, the logarithmic growth dominates:

$$\lim_{n \to \infty} \frac{100 \log_2 n}{n} = 0$$

This means that for sufficiently large n, $O(\log n)$ algorithms **win** over linear algorithms despite higher per-step cost.

—

2. Effect of Lower-Order Terms

Consider two functions:

$$f(n) = n^3, \quad g(n) = n^3 + 10n^2$$

Absolute Difference

The absolute difference is

$$|g(n) - f(n)| = |(n^3 + 10n^2) - n^3| = 10n^2$$

For small n, this might be significant. For large n, $10n^2 \ll n^3$, so the additional term is negligible in the absolute sense.

Relative Difference

The relative difference is

$$\frac{|g(n) - f(n)|}{f(n)} = \frac{10n^2}{n^3} = \frac{10}{n}$$

As $n \to \infty$, the relative difference $\to 0$. This illustrates a key principle in algorithm analysis:

> *Lower-order terms and constant factors typically do not affect the asymptotic growth rate.*

Hence, $f(n)$ and $g(n)$ are both $O(n^3)$, and for large input sizes, the $10n^2$ term becomes negligible.

—

3. Summary

- Even if an algorithm has more expensive steps, a slower-growing asymptotic complexity (like $O(\log n)$) can outperform a higher-growth complexity (like $O(n)$) for sufficiently large n.
- Lower-order terms and constants often do not matter asymptotically. Absolute differences may exist, but relative differences vanish as n becomes large.