Functions and Their Growth Rates

1 Mathematical Functions in Algorithm Analysis

In computer science, particularly in the analysis of algorithms, we use mathematical functions to describe how the resources required by an algorithm grow as the input size increases. The input size is usually denoted by n. The resources we are most concerned with are time (how long the algorithm takes to run) and space (how much memory it uses).

A function $f(n)$ maps the input size n to the number of steps or operations the algorithm performs. By studying these functions, we can predict whether an algorithm will be efficient enough to handle large datasets. We are not usually interested in the exact number of seconds, but rather the ”shape” of the curve that the function draws. This shape tells us the growth rate.

2 Common Types of Functions

There are several standard classes of functions that appear frequently when analyzing algorithms. Understanding these is essential for comparing efficiency.

2.1 Constant Functions

A constant function is written as $f(n) = c$, where c is a fixed number. For example, $f(n) = 1$ or $f(n) = 100$.

- **Meaning:** The running time does not change, regardless of the input size.
- **Example:** Accessing a specific element in an array by its index takes constant time.

2.2 Logarithmic Functions

A logarithmic function is written as $f(n) = \log_b n$. In computer science, the base b is usually 2, so we often just write $\log n$.

- **Meaning:** The function grows very slowly. If the input size doubles, the number of operations increases by only a small, constant amount.

- **Example:** Binary search algorithms, where the problem size is cut in half at each step.

2.3 Linear Functions

A linear function is written as $f(n) = an + b$, or simply proportional to n.

- **Meaning:** The running time grows directly in proportion to the input. If you double the input, the time doubles.
- **Example:** Iterating through a list of items once, such as finding the maximum value in an unsorted array.

2.4 Linearithmic Functions

A linearithmic function is written as $f(n) = n\log n$.

- **Meaning:** Thisgrowsslightlyfasterthanalinearfunctionbutmuchslowerthanaquadratic function. It represents the combination of a linear pass and a logarithmic process.
- **Example:** Efficient sorting algorithms like Merge Sort and Quick Sort.

2.5 Polynomial Functions

Polynomial functions include quadratic (n^2), cubic (n^3), and higher powers.

- **Quadratic (n^2):** Commonly seen in algorithms with nested loops (a loop inside a loop). Example: Bubble Sort.
- **Cubic(n^3):** Seen in algorithms with triple nested loops, such as naive matrix multiplication.

2.6 Exponential Functions

An exponential function is written as $f(n) = b^n$, where $b > 1$. Common bases are 2^n or 10^n.

- **Meaning:** These functions grow extremely fast. Even for small values of n, the result becomes huge.
- **Example:** Solving the Traveling Salesperson Problem using a brute-force approach.

2.7 Factorial Functions

A factorial function is written as $f(n) = n!$, which is $n \times (n-1) \times \cdots \times 1$.

- **Meaning:** This is the fastest growing function among the common types. It usually represents generating all possible permutations of a set.
- **Example:** Finding all possible arrangements of a list of items.

3 Growth Rates of Functions

The growth rate of a function refers to the rate at which $f(n)$ increases as n approaches infinity. When n is small (e.g., $n = 10$), the difference between n^2 and n^3 might not seem significant. However, as n becomes very large (e.g., $n = 1{,}000{,}000$), the difference in growth rates becomes the deciding factor in whether an algorithm is usable.

We study growth rates to classify algorithms into efficiency tiers. An algorithm with a lower growth rate is considered more efficient for large inputs.

4 Comparison of Function Growth

To evaluate algorithms, we compare their growth functions.

4.1 Ordering of Common Functions

The following list orders common functions from the slowest growth (most efficient) to the fastest growth (least efficient).

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$$

This inequality holds for sufficiently large values of n.

4.2 Dominant Terms in Functions

When a function consists of multiple terms, the term with the highest growth rate is called the *dominant term*. As n becomes large, the dominant term accounts for the vast majority of the value.

For example, consider the function $f(n) = 2n^2 + 100n + 500$.

- As $n \to \infty$, the term $2n^2$ becomes much larger than $100n$ or 500.

- We say that n^2 dominates the function.
- Therefore, the growth rate is determined by n^2.

5 Properties of Growth Rates

When analyzing growth rates using asymptotic notation (like Big-O), we apply two main simplification rules.

5.1 Ignoring Constants

Multiplicative constants do not change the fundamental class of the growth rate.

- The functions n, $5n$, and $100n$ all have linear growth.
- Although $100n$ is larger than n, they both grow linearly. We treat them as the same class of growth: $O(n)$.

5.2 Ignoring Lower-Order Terms

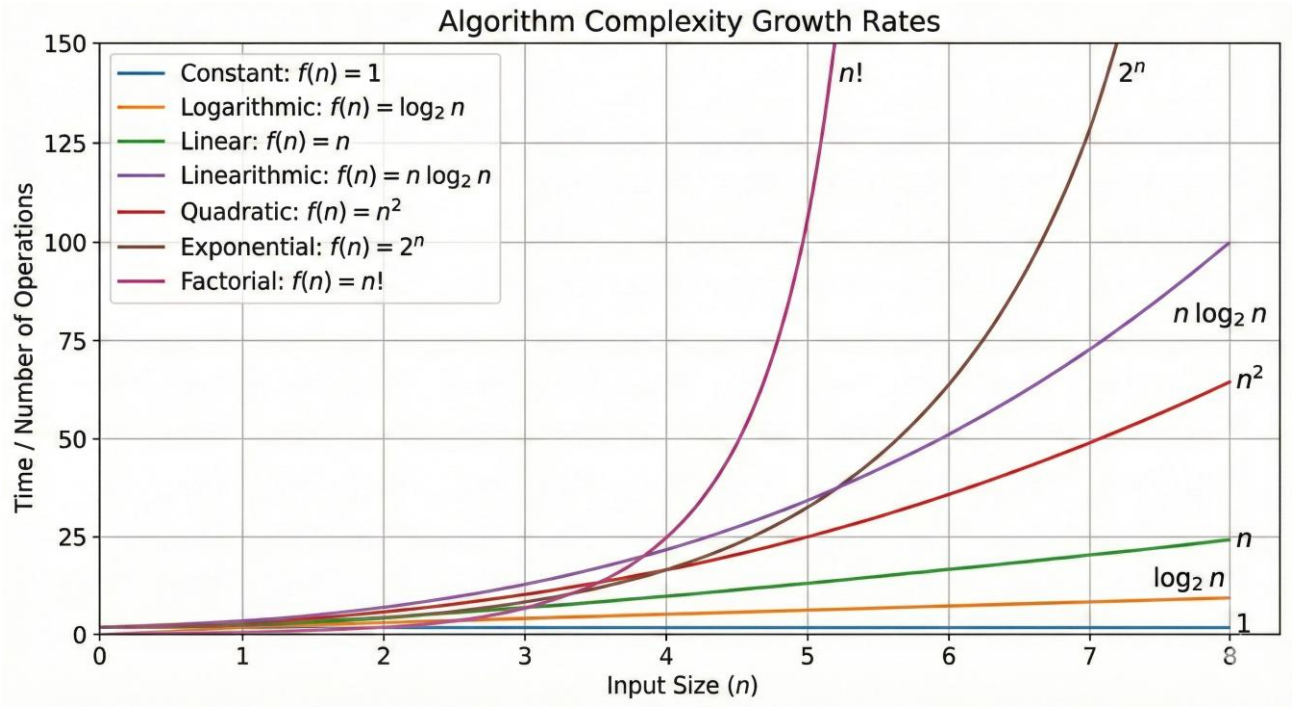
We only keep the term that grows the fastest. Lower-order terms become insignificant for large inputs.

- In the function $f(n) = n^3 + n^2 + n$, the term n^3 is the highest order.
- We simplify this to just n^3 when discussing growth rate.

6 Graphical Intuition of Function Growth

If we were to plot these functions on a graph where the x-axis is n (input size) and the y-axis is time:

- The **Constant** line (1) is flat.
- The **Logarithmic** line $(\log n)$ rises initially but flattens out very quickly.
- The **Linear** line (n) is a straight diagonal line.
- The **Quadratic** line (n^2) curves upward in a parabola.
- The **Exponential** line (2^n) stays low for small n, then shoots upward almost vertically, representing a ”wall” where the problem becomes unsolvable for slightly larger inputs.



7 Practical Significance of Growth Rates

Understanding growth rates is not just a theoretical exercise; it has real-world consequences for software development.

1. **Scalability:** An algorithm with $O(n^2)$ might work fine for 100 users, but if the system grows to 1,000,000 users, the server may crash or become unresponsive. An $O(n\log n)$ algorithm would likely handle the load.
2. **Hardware Limits:** You cannot simply buy a faster computer to fix a bad growth rate. If an algorithm is exponential (2^n), doubling the processor speed might only allow you to increase the input size n by 1.
3. **Battery Life:** On mobile devices, inefficient algorithms (high growth rates) consume more CPU cycles, which drains the battery faster.

8 Summary of Key Concepts

- We analyze functions to predict algorithm performance on large inputs.
- Common growth classes include Constant, Logarithmic, Linear, Linearithmic, Quadratic, and Exponential.
- When comparing functions, we focus on the dominant term and ignore constants.

- The hierarchy $1 < \log n < n < n\log n < n^2 < 2^n$ is the standard reference for efficiency.

9 Practice Problems and Exercises

1. Arrange the following functions in increasing order of growth rate:

$$n! \quad n \quad \sqrt{n} \quad n^2 \quad \log(n!)$$

2. Simplify the following functions to their dominant growth term:
   - $f(n) = 3n^4 + 10n^2 + 5$
   - $g(n) = 5 \cdot 2^n + n^{10}$
3. Why is an algorithm with growth rate $O(n\log n)$ preferred over $O(n^2)$ for sorting large databases?
4. If an algorithm takes n^2 operations and your computer can perform 10^9 operations per second, roughly how long will it take to process an input of size $n = 100{,}000$?