Algorithm Analysis & Asymptotic Notation

Data Structures and Algorithms

Masoud Hamad

State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

1. Why Analyze Algorithms?
2. Asymptotic Notation
   - Big-O Notation
   - Big-Omega Notation
   - Big-Theta Notation
   - Little-o and Little-omega
3. Common Complexity Classes
4. Asymptotic Operations & Properties
5. Analyzing Code Complexity
6. Recurrence Relations
7. Best, Average, and Worst Case
8. Space Complexity
9. Summary & Practice

Why Analyze Algorithms?

Two programs can solve the same problem, but one may be **much faster** or use **much less memory**.

Algorithm Analysis answers:

- How does the running time grow as input size increases?
- How much memory does the algorithm need?
- Which algorithm is better for a given problem?

We need a way to compare algorithms **independently** of hardware, programming language, or compiler.

Example

Searching in an array of n elements:

- Linear Search: checks up to n elements
- Binary Search: checks up to $\log_2 n$ elements

For $n = 1{,}000{,}000$:
Linear: 1,000,000 steps
Binary: $\approx$ 20 steps

Measuring Efficiency

Bad Approach: Wall Clock Time

- Depends on computer speed
- Depends on programming language
- Depends on other running programs
- Not reproducible

Good Approach: Count Operations

- Count basic operations (comparisons, assignments, arithmetic)
- Express as a function of input size n
- Focus on growth rate as $n \to \infty$
- Independent of hardware

Key Idea

We care about **how fast** the running time **grows** with input size, not the exact number of operations.

Counting Operations — Example

```
int sum(int arr[], int n) {
    int total = 0;           // 1 operation
    for (int i = 0; i < n; i++) {
        // loop runs n times
        total += arr[i];     // 1 op * n
            times
    }
    return total;            // 1 operation
}
```

Total operations: $1 + n + 1 = n + 2$

As n grows large, the "+2" becomes insignificant.
We say this is $O(n)$.

```
void printPairs(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d,%d\n",
                   arr[i], arr[j]);
        }
    }
}
```

Inner loop runs n times for each of n outer iterations.

Total: $n \times n = n^2$ operations.
We say this is $O(n^2)$.

Asymptotic Notation — Overview

Asymptotic notation describes how a function grows as $n \to \infty$.

Notation	Name	Meaning
$O(g(n))$	Big-O	Upper bound (at most)
$\Omega(g(n))$	Big-Omega	Lower bound (at least)
$\Theta(g(n))$	Big-Theta	Tight bound (exactly)
$o(g(n))$	Little-o	Strictly less than
$\omega(g(n))$	Little-omega	Strictly more than

Analogy

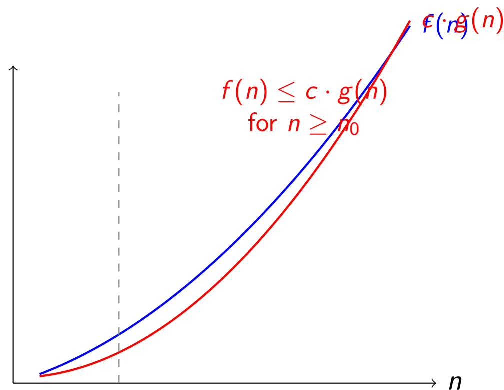
Think of comparing two numbers a and b:
O is like $\leq$, Ω is like $\geq$, Θ is like $=$, o is like $<$, ω is like $>$

Big-O Notation (O) — Upper Bound

Formal Definition

$f(n) = O(g(n))$ if there exist positive constants c and n_0 such that:

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$



Big-O — Examples

Example 1: $f(n) = 3n + 5$ is $O(n)$

Choose $c = 4$, $n_0 = 5$:
$3n + 5 \leq 4n$ for all $n \geq 5$ ✓

Example 2: $f(n) = 2n^2 + 3n + 1$ is $O(n^2)$

Choose $c = 3$, $n_0 = 4$:
$2n^2 + 3n + 1 \leq 3n^2$ for all $n \geq 4$ ✓
(since $3n + 1 \leq n^2$ for $n \geq 4$)

Example 3: $f(n) = n^2$ is **NOT** $O(n)$

No matter what c we choose, $n^2 > cn$ for large enough n.
(divide both sides by n: $n > c$, which is true for $n > c$)

Rule of Thumb

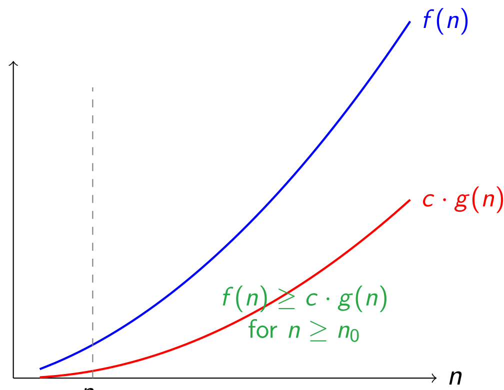
Drop lower-order terms and constant coefficients:

Big-Omega Notation (Ω) — Lower Bound

Formal Definition

$f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that:

$$0 \leq c \cdot g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

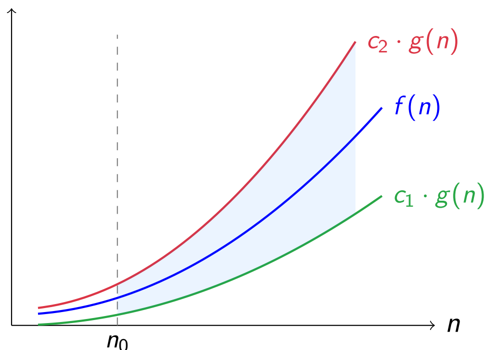


Big-Theta Notation (Θ) — Tight Bound

Formal Definition

$f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0$$



Big-Theta — Examples

Example: $f(n) = 2n^2 + 3n + 1$ is $\Theta(n^2)$

Upper bound (O): $2n^2 + 3n + 1 \leq 3n^2$ for $n \geq 4$ $\quad (c_2 = 3)$
Lower bound (Ω): $2n^2 + 3n + 1 \geq 2n^2$ for $n \geq 1$ $\quad (c_1 = 2)$

Therefore: $2n^2 \leq 2n^2 + 3n + 1 \leq 3n^2$ for $n \geq 4$
So $f(n) = \Theta(n^2)$ ✓

Function	$O(\cdot)$	$\Omega(\cdot)$	$\Theta(\cdot)$
$5n + 3$	$O(n)$	$\Omega(n)$	$\Theta(n)$
$2n^2 + n$	$O(n^2)$	$\Omega(n^2)$	$\Theta(n^2)$
n^2	$O(n^2)$, $O(n^3)$	$\Omega(n)$, $\Omega(n^2)$	$\Theta(n^2)$
$3 \cdot 2^n + n^3$	$O(2^n)$	$\Omega(2^n)$	$\Theta(2^n)$

Note: Θ gives the **tightest** characterization. O can be loose (e.g., $n = O(n^{100})$ is true but useless).

Little-o and Little-ω Notation

Little-o: Strictly Less Than

$f(n) = o(g(n))$ means: for **every** constant $c > 0$, there exists n_0 such that:

$$0 \leq f(n) < c \cdot g(n) \quad \text{for all } n \geq n_0$$

Equivalently: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

Little-ω: Strictly Greater Than

$f(n) = \omega(g(n))$ means: for **every** constant $c > 0$, there exists n_0 such that:

$$0 \leq c \cdot g(n) < f(n) \quad \text{for all } n \geq n_0$$

Equivalently: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Summary of All Asymptotic Notations

Notation	Analogy	Limit	Informal Meaning
$f = O(g)$	$a \leq b$	$\limsup \frac{f}{g} < \infty$	f grows at most as fast as g
$f = \Omega(g)$	$a \geq b$	$\liminf \frac{f}{g} > 0$	f grows at least as fast as g
$f = \Theta(g)$	$a = b$	$0 < \lim \frac{f}{g} < \infty$	f grows at same rate as g
$f = o(g)$	$a < b$	$\lim \frac{f}{g} = 0$	f grows strictly slower than g
$f = \omega(g)$	$a > b$	$\lim \frac{f}{g} = \infty$	f grows strictly faster than g

Key Relationships

- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$
- $f(n) = o(g(n)) \implies f(n) = O(g(n))$, but not vice versa
- $f(n) = \omega(g(n)) \implies f(n) = \Omega(g(n))$, but not vice versa

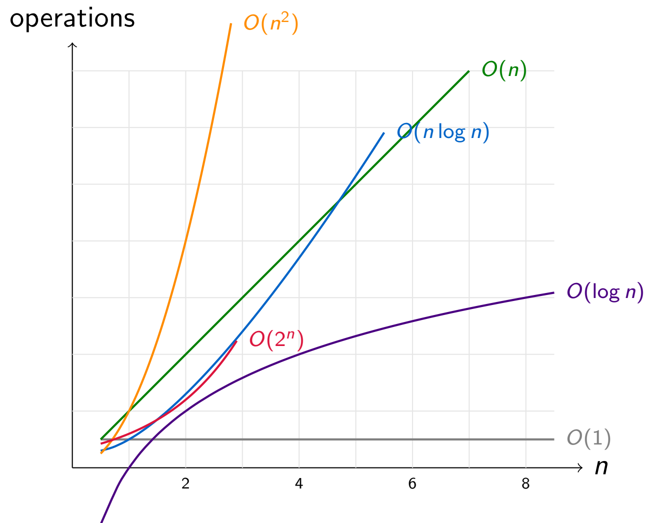
Common Complexity Classes

Notation	Name	Example	$n = 10^6$
$O(1)$	Constant	Array access	1
$O(\log n)$	Logarithmic	Binary search	20
$O(n)$	Linear	Linear search	10^6
$O(n \log n)$	Log-linear	Merge sort	2×10^7
$O(n^2)$	Quadratic	Bubble sort	10^{12}
$O(n^3)$	Cubic	Matrix multiply	10^{18}
$O(2^n)$	Exponential	Subsets	$\approx 10^{301029}$
$O(n!)$	Factorial	Permutations	$\approx 10^{5565709}$

Ordering:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!)$$

Growth Rates — Visual Comparison



Practical Rule

Asymptotic Operations — Addition

Rule: Sum of Functions

If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then:

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

The dominant term wins!

Examples

- $O(n) + O(n^2) = O(n^2)$
- $O(n \log n) + O(n) = O(n \log n)$
- $O(n^3) + O(2^n) = O(2^n)$

Application: Sequential code blocks — total time is the **maximum** of individual blocks.

$$\text{Block 1: } O(n) \;+\; \text{Block 2: } O(n^2) \;+\; \text{Block 3: } O(n) \;=\; O(n^2)$$

Asymptotic Operations — Multiplication

Rule: Product of Functions

If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then:

$$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

Examples

- $O(n) \cdot O(n) = O(n^2)$ (nested loops)
- $O(\log n) \cdot O(n) = O(n \log n)$ (e.g., merge sort)
- $O(n) \cdot O(1) = O(n)$ (constant work inside loop)

Constants Can Be Dropped

$c \cdot O(f(n)) = O(f(n))$ for any constant $c > 0$

Examples: $5 \cdot O(n) = O(n)$, $100 \cdot O(n^2) = O(n^2)$

Properties of Asymptotic Notation

1. Transitivity

If $f = O(g)$ and $g = O(h)$,
then $f = O(h)$.

Same for Ω, Θ, o, ω.

2. Reflexivity

$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$
$f(n) = \Theta(f(n))$

3. Symmetry

$f = \Theta(g) \iff g = \Theta(f)$

Only Θ *is symmetric!*

4. Transpose Symmetry

$f = O(g) \iff g = \Omega(f)$

$f = o(g) \iff g = \omega(f)$

5. Polynomial Dominance

For $a > b > 0$:
$n^b = o(n^a)$

Example: $n^2 = o(n^3)$

6. Exponential Dominance

For any $k > 0$ and $a > 1$:
$n^k = o(a^n)$

Example: $n^{100} = o(2^n)$

Useful Limit Rules for Asymptotic Analysis

Using Limits to Determine Relationship

Given $L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$:

$$
\begin{aligned}
&\text{If } L = 0 && \Rightarrow \quad f(n) = o(g(n)) \text{ and } f(n) = O(g(n)) \\
&\text{If } 0 < L < \infty && \Rightarrow \quad f(n) = \Theta(g(n)) \\
&\text{If } L = \infty && \Rightarrow \quad f(n) = \omega(g(n)) \text{ and } f(n) = \Omega(g(n))
\end{aligned}
$$

Examples

$$\lim_{n \to \infty} \frac{n}{\log n} = \infty \quad \Rightarrow \quad n = \omega(\log n), \text{ so } \log n = o(n)$$

$$\lim_{n \to \infty} \frac{3n^2 + 5n}{n^2} = 3 \quad \Rightarrow \quad 3n^2 + 5n = \Theta(n^2)$$

$$\lim \frac{n^2}{2^n} = 0 \quad \Rightarrow \quad n^2 = o(2^n)$$

Rule 1: Sequential Statements

```
int a = 5;                    // O(1)
int b = a + 10;               // O(1)
printf("%d\n", b);            // O(1)
```

Total: $O(1) + O(1) + O(1) = O(1)$

```
// Block 1: O(n)
for (int i = 0; i < n; i++)
    printf("%d ", arr[i]);

// Block 2: O(n^2)
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        printf("%d,%d ", arr[i], arr[j]);
```

Total: $O(n) + O(n^2) = O(n^2)$

Rule

For sequential blocks, take the **maximum**: $O(\max(f_1, f_2, \ldots))$

Rule 2: Simple Loops

```
// Runs n times -> O(n)
for (int i = 0; i < n; i++) {
    // O(1) work
}

// Runs n/2 times -> O(n)
for (int i = 0; i < n; i += 2) {
    // O(1) work
}

// Runs n-5 times -> O(n)
for (int i = 5; i < n; i++) {
    // O(1) work
}
```

```
// Logarithmic: O(log n)
for (int i = 1; i < n; i *= 2) {
    // O(1) work
}
// i = 1, 2, 4, 8, ..., n
// Steps: log2(n)

// Logarithmic: O(log n)
for (int i = n; i > 0; i /= 2) {
    // O(1) work
}
// i = n, n/2, n/4, ..., 1
// Steps: log2(n)
```

Rule

- Loop variable incremented by constant $\rightarrow O(n)$
- Loop variable multiplied/divided by constant $\rightarrow O(\log n)$

Rule 3: Nested Loops

```
// O(n) * O(n) = O(n^2)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // O(1) work
    }
}
```

```
// O(n) * O(n) * O(n) = O(n^3)
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      // O(1) work
```

```
// Dependent loops: O(n^2)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        // O(1) work
    }
}
// Total: 0+1+2+...+(n-1)
//        = n(n-1)/2 = O(n^2)
```

```
// O(n) * O(log n) = O(n log n)
for (int i = 0; i < n; i++) {
    for (int j = 1; j < n; j *= 2){
        // O(1) work
    }
}
```

Rule

Multiply the complexities of each loop level.

Rule 4: If-Else Statements

```
if (condition) {
    // Block A: O(n)
    for (int i = 0; i < n; i++)
        process(arr[i]);
} else {
    // Block B: O(n^2)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            compare(arr[i], arr[j]);
}
```

Rule

For if-else, take the **worst case** (maximum):

$$O(\max(n, n^2)) = O(n^2)$$

We analyze the worst case — the branch that takes the most time.

Rule 5: Recursive Functions

```
// Linear recursion: O(n)
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
// T(n) = T(n-1) + O(1)
// T(n) = O(n)
```

```
// Binary recursion: O(2^n)
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
// T(n) = T(n-1) + T(n-2) + O(1)
// T(n) = O(2^n)
```

```
// Divide and conquer: O(n log n)
void mergeSort(int arr[],
               int l, int r) {
    if (l >= r) return;
    int m = (l + r) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m+1, r);
    merge(arr, l, m, r); // O(n)
}
// T(n) = 2T(n/2) + O(n)
// T(n) = O(n log n)
```

```
// Binary search: O(log n)
// T(n) = T(n/2) + O(1)
// T(n) = O(log n)
```

Recurrence Relations

What is a Recurrence?

A recurrence relation expresses the running time of a recursive algorithm in terms of smaller inputs.

Algorithm	**Recurrence**	**Solution**
Linear Search	$T(n) = T(n-1) + O(1)$	$O(n)$
Binary Search	$T(n) = T(n/2) + O(1)$	$O(\log n)$
Merge Sort	$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
Quick Sort (avg)	$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
Quick Sort (worst)	$T(n) = T(n-1) + O(n)$	$O(n^2)$
Fibonacci (naive)	$T(n) = T(n-1) + T(n-2) + O(1)$	$O(2^n)$

Methods to Solve:

1. Substitution (expand and find pattern)
2. Recursion Tree

Solving by Expansion (Substitution)

Example: $T(n) = T(n/2) + 1, \quad T(1) = 1$

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= [T(n/4) + 1] + 1 = T(n/4) + 2 \\ &= [T(n/8) + 1] + 2 = T(n/8) + 3 \\ &\vdots \\ &= T(n/2^k) + k \end{aligned}$$

Base case when $n/2^k = 1 \Rightarrow k = \log_2 n$:

$$T(n) = T(1) + \log_2 n = 1 + \log_2 n = O(\log n)$$

Verification

Binary search: $T(n) = T(n/2) + O(1) = O(\log n)$ ✓

Master Theorem

For Recurrences of the Form

$$T(n) = aT(n/b) + O(n^d)$$

where $a \geq 1,\ b > 1,\ d \geq 0$

Case	**Solution**
Case 1: $d < \log_b a$	$T(n) = O(n^{\log_b a})$
Case 2: $d = \log_b a$	$T(n) = O(n^d \log n)$
Case 3: $d > \log_b a$	$T(n) = O(n^d)$

Examples

- Merge Sort: $T(n) = 2T(n/2) + O(n) \rightarrow a = 2, b = 2, d = 1 \rightarrow d = \log_2 2 = 1 \rightarrow$ Case 2: $O(n \log n)$

Best, Average, and Worst Case

An algorithm's performance can vary depending on the **input**:

Three Cases

- **Best Case:** Minimum operations (most favorable input)
- **Average Case:** Expected operations (typical input)
- **Worst Case:** Maximum operations (most unfavorable input)

We usually analyze the **worst case** because it gives a **guarantee**.

Insertion Sort

Best:	$O(n)$ (sorted input)
Average:	$O(n^2)$
Worst:	$O(n^2)$ (reverse sorted)

Quick Sort

Best:	$O(n \log n)$
Average:	$O(n \log n)$
Worst:	$O(n^2)$ (sorted + bad pivot)

Space Complexity

Definition

Space complexity measures the **total memory** used by an algorithm as a function of input size, including:

- **Input space:** memory for input data
- **Auxiliary space:** extra memory used during execution

```c
// O(1) auxiliary space
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
// O(n) auxiliary space
int *copy = malloc(n * sizeof(int));
for (int i = 0; i < n; i++)
    copy[i] = arr[i];
```

Algorithm	**Space**
Bubble Sort	$O(1)$
Selection Sort	$O(1)$
Insertion Sort	$O(1)$
Merge Sort	$O(n)$
Quick Sort	$O(\log n)$
Binary Search	$O(1)$ iter
Binary Search	$O(\log n)$ rec

Summary

Key Takeaways

1. **Algorithm analysis** measures efficiency independent of hardware
2. **Big-O** (O): upper bound — "at most this fast"
3. **Big-Omega** (Ω): lower bound — "at least this fast"
4. **Big-Theta** (Θ): tight bound — "exactly this fast"
5. **Little-o/omega**: strict (non-tight) bounds
6. **Drop constants and lower-order terms** in asymptotic analysis
7. **Sequential code**: take the maximum
8. **Nested loops**: multiply the complexities
9. **Recursion**: write and solve recurrence relations
10. **Master Theorem**: $T(n) = aT(n/b) + O(n^d)$

Practice Problems

Determine the Big-O complexity:

1. $f(n) = 5n^3 + 2n^2 + 100$
2. $f(n) = n \log n + n$
3. $f(n) = 3^n + n^{10}$
4. A loop from 1 to n with a nested loop from 1 to $\sqrt{n}$
5. $T(n) = 3T(n/3) + O(n)$ (use Master Theorem)
6. $T(n) = 4T(n/2) + O(n)$ (use Master Theorem)

Prove or disprove:

7. $2n + 10 = O(n)$
8. $n^2 = O(n)$
9. $n \log n = \Omega(n)$
10. $n^2 + n = \Theta(n^2)$

Questions?