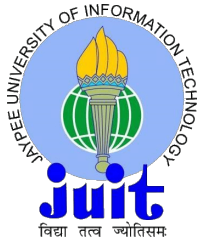


Data Structure

(Course Code: 25B11CI318)

Asymptotic Notations

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Algorithm Analysis

Definition

Algorithm Analysis is the process of determining the resources required by an algorithm as a function of the input size.

The two primary resources are:

1. **Time** – How much computational work is performed?
2. **Space** – How much memory is required?

Goal

Compare algorithms and determine which algorithm is more efficient for a given problem.

Why Analyze Algorithms?

Consider two algorithms for the same problem.

$$T_1(n) = 100n$$

$$T_2(n) = n^2$$

For small n, both may appear efficient.

Why Analyze Algorithms?

Consider two algorithms for the same problem.

$$T_1(n) = 100n$$

$$T_2(n) = n^2$$

For small n, both may appear efficient.
However, as n increases:

$$100n \ll n^2$$

- Algorithm analysis helps predict performance.
- It allows us to compare algorithms independently of hardware.
- It helps select an appropriate algorithm for large inputs.

Key Idea

Input Size

The efficiency of an algorithm depends on the size of its input.

Let:

$$n = \text{size of the input}$$

Examples:

- Array of n elements $\rightarrow$ n is the number of elements.
- String of n characters $\rightarrow$ n is the string length.
- Graph $\rightarrow$ n may represent number of vertices.
- Matrix $\rightarrow$ n may represent number of rows/columns.

Important

The definition of input size depends on the problem being solved.

Time Complexity

Definition

Time Complexity describes how the running time of an algorithm grows as a function of the input size n.

We express running time as:

$$T(n)$$

where n is the input size.

Examples:

$$T(n) = 5$$

$$T(n) = 3n + 2$$

Counting Basic Operations

Consider:

```
sum = a + b;
```

The statement performs a constant amount of work.

$$T(n) = c$$

Therefore:

$$\boxed{T(n) = O(1)}$$

For:

```
for(i = 0; i < n; i++)
{
    sum = sum + a[i];
}
```

Example: Constant Time

```
int first = A[0];
```

Only one array element is accessed. Number of operations does not depend on n.

$$T(n) = c$$

Therefore:

$$\boxed{T(n) = O(1)}$$

Examples

- Accessing an array element
- Assigning a value
- Arithmetic operation
- Comparing two values

Example: Linear Time

```
for(i = 0; i < n; i++)
{
    printf("%d", A[i]);
}
```

The loop executes:

$$n \text{ times}$$

Therefore:

$$T(n) = cn$$

$$\boxed{T(n) = O(n)}$$

Pattern

One loop that runs from 0 to $n-1$ usually gives linear complexity.

Example: Quadratic Time

```
for(i = 0; i < n; i++)
{
    for(j = 0; j < n; j++)
    {
        printf("*");
    }
}
```

Outer loop:

n iterations

Inner loop:

n iterations for each outer iteration

Total:

$$n \times n = n^2$$

Therefore:

$$\boxed{T(n) = O(n^2)}$$

Space Complexity

Definition

Space Complexity is the amount of memory required by an algorithm as a function of the input size.

Space can include:

- Input storage
- Variables
- Auxiliary data structures
- Function call stack
- Dynamically allocated memory

Example:

```
int sum;
```

Only a constant amount of extra memory is required.

$$S(n) = O(1)$$

Time Complexity vs Space Complexity

Time Complexity	**Space Complexity**
Measures computational work	Measures memory usage
Depends on input size	Depends on input size
Represented by $T(n)$	Represented by $S(n)$
Example: $O(n)$	Example: $O(n)$

Trade-off

Sometimes we use more memory to reduce running time.

$$\boxed{\text{More Space} \Longleftrightarrow \text{Less Time}}$$

Best, Average and Worst Case

An algorithm may behave differently for different inputs of the same size.

Best Case: Minimum time required.

Average Case: Expected time over typical inputs.

Worst Case: Maximum time required.

Example: Linear Search

Search:

$$10,\ 20,\ 30,\ 40,\ 50$$

Best, Average and Worst Case

An algorithm may behave differently for different inputs of the same size.

Best Case: Minimum time required.

Average Case: Expected time over typical inputs.

Worst Case: Maximum time required.

Example: Linear Search
Search:

$$10,\ 20,\ 30,\ 40,\ 50$$

Searching for:

- 10 $\rightarrow$ Best Case
- 30 $\rightarrow$ Average Case
- 50 $\rightarrow$ Worst Case

Linear Search: Complexity

```
for(i = 0; i < n; i++)
{
    if(A[i] == key)
        return i;
}
```

Best Case:

$$\boxed{O(1)}$$

Element is found at the first position.

Linear Search: Complexity

```
for(i = 0; i < n; i++)
{
    if(A[i] == key)
        return i;
}
```

Best Case:

$$\boxed{O(1)}$$

Element is found at the first position.

Worst Case:

$$\boxed{O(n)}$$

Element is at the last position or absent.

Linear Search: Complexity

```
for(i = 0; i < n; i++)
{
    if(A[i] == key)
        return i;
}
```

Best Case:

$$\boxed{O(1)}$$

Element is found at the first position.

Worst Case:

$$\boxed{O(n)}$$

Element is at the last position or absent.

Average Case:

$$\boxed{O(n)}$$

Asymptotic Analysis

Definition

Asymptotic analysis studies the growth of an algorithm's running time as the input size n becomes very large.

It ignores:

- Machine-dependent constants
- Programming language details
- Lower-order terms

For example:

$$T(n) = 3n^2 + 5n + 10$$

For large n:

$$T(n) \approx 3n^2$$

Ignoring the constant:

Why Ignore Constants?

Consider:

$$T_1(n) = 10n$$

and

$$T_2(n) = 100n$$

Both have:

$$\boxed{O(n)}$$

The constants may affect actual running time, but they do not change the growth rate.

Why Ignore Constants?

Consider:

$$T_1(n) = 10n$$

and

$$T_2(n) = 100n$$

Both have:

$$\boxed{O(n)}$$

The constants may affect actual running time, but they do not change the growth rate.
Similarly:

$$T(n) = 5n^2 + 20n + 100$$

hog:

Big-O Notation

Definition

Big-O notation provides an **asymptotic upper bound** on the growth of a function.

We write:

$$f(n) = O(g(n))$$

if $f(n)$ does not grow faster than a constant multiple of $g(n)$ for sufficiently large n.

Examples:

$$3n + 5 = O(n)$$

$$2n^2 + 3n + 1 = O(n^2)$$

Formal Definition of Big-O

We say:

$$f(n) = O(g(n))$$

if there exist positive constants c and n_0 such that:

$$0 \leq f(n) \leq c\, g(n)$$

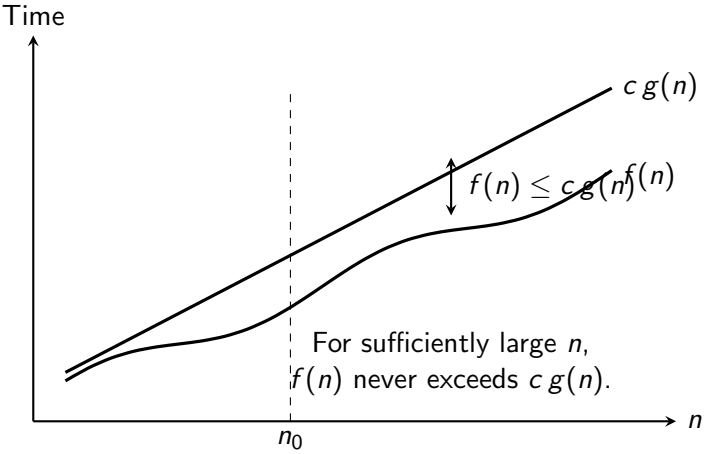
for all:

$$n \geq n_0$$

Interpretation

For sufficiently large n, $f(n)$ is bounded above by a constant multiple of $g(n)$.

Big-O Notation



$$f(n) = O(g(n))$$

Big-Ω Notation

Definition

Big-Ω notation provides an **asymptotic lower bound** on the growth of a function.

We write:

$$f(n) = \Omega(g(n))$$

if there exist positive constants c and n_0 such that:

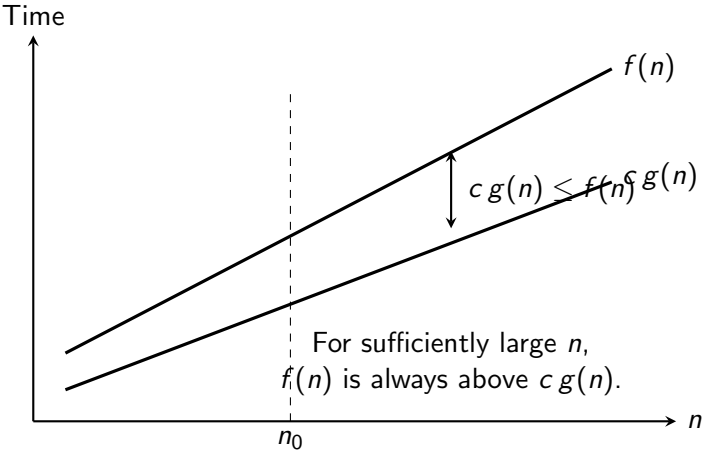
$$0 \leq c\, g(n) \leq f(n)$$

for all:

$$n \geq n_0$$

Example:

Big-Ω Notation



$$f(n) = \Omega(g(n))$$

Big-Θ Notation

Definition

Big-Θ notation provides a **tight asymptotic bound**.

We write:

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

if there exist positive constants c_1, c_2, n_0 such that:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

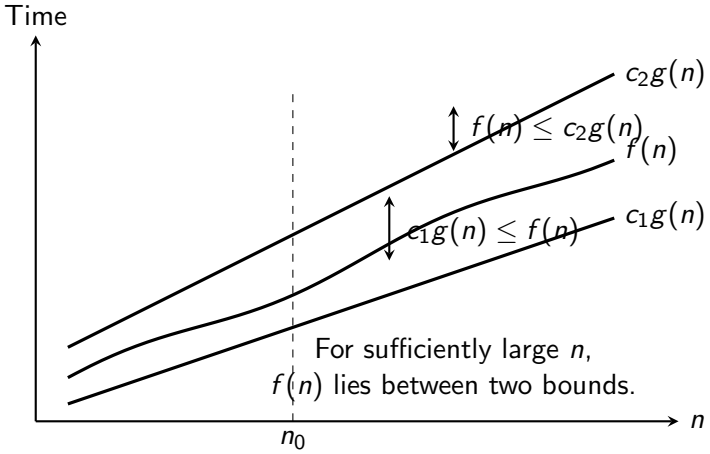
for all:

$$n \geq n_0$$

Example:

$$3n^2 + 5n + 2 = \Theta(n^2)$$

Big-Θ Notation



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

Big-O, Big-Ω and Big-Θ

Notation	Meaning	Bound
$O(g(n))$	Upper bound	At most
$\Omega(g(n))$	Lower bound	At least
$\Theta(g(n))$	Tight bound	Both

For:

$$f(n) = 3n^2 + 5n + 10$$

we can say:

$$f(n) = O(n^2)$$

$$f(n) = \Omega(n^2)$$

$$\boxed{f(n) = \Theta(n^2)}$$

Common Asymptotic Growth Rates

From slower growth to faster growth:

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

Complexity	Common Example
$O(1)$	Array access
$O(\log n)$	Binary Search
$O(n)$	Linear Search
$O(n \log n)$	Merge Sort
$O(n^2)$	Bubble Sort
$O(2^n)$	Recursive subset generation
$O(n!)$	Permutation generation

Growth Rate: Intuition

For large values of n:

$$O(1)$$

remains almost constant.

$$O(\log n)$$

grows very slowly.

$$O(n)$$

grows linearly.

$$O(n^2)$$

grows much faster.

$$O(2^n)$$

becomes impractical very quickly.

Key Idea

Asymptotic analysis helps us understand which algorithms remain practical as input size increases.

Rule 1: Drop Constants

Consider:

$$T(n) = 5n$$

Ignoring the constant:

$$\boxed{O(n)}$$

Another example:

$$T(n) = 100n + 20$$

Therefore:

$$\boxed{O(n)}$$

Rule

Constant multipliers are ignored in asymptotic notation.

Rule 2: Drop Lower-Order Terms

Suppose:

$$T(n) = 3n^2 + 5n + 10$$

The dominant term is:

$$n^2$$

Therefore:

$$\boxed{T(n) = O(n^2)}$$

Similarly:

$$7n^3 + 4n^2 + 10n + 5$$

becomes:

$$\boxed{O(n^3)}$$

Rule 3: Sequential Statements

```
for(i = 0; i < n; i++)
{
    statement1;
}

for(i = 0; i < n; i++)
{
    statement2;
}
```

First loop:

$$O(n)$$

Second loop:

$$O(n)$$

Total:

$$O(n) + O(n) = O(2n)$$

Therefore:

$$\boxed{O(n)}$$

Rule 4: Nested Loops

```
for(i = 0; i < n; i++)
{
    for(j = 0; j < n; j++)
    {
        statement;
    }
}
```

Outer loop:

$$n$$

Inner loop:

$$n$$

Therefore:

$$n \times n = n^2$$

$$\boxed{O(n^2)}$$

Rule 5: Dependent Nested Loops

```
for(i = 0; i < n; i++)
{
    for(j = 0; j < i; j++)
    {
        statement;
    }
}
```

Number of executions:

$$0 + 1 + 2 + \cdots + (n-1)$$

Using:

$$\frac{n(n-1)}{2}$$

Therefore:

$$T(n) = \frac{n(n-1)}{2}$$

Ignoring constants and lower-order terms:

$$\boxed{O(n^2)}$$

Rule 6: If-Else Statements

```
if(condition)
{
    // O(n)
}
else
{
    // O(n^2)
}
```

Only one branch executes. Therefore, take the larger complexity:

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

$$\boxed{O(n^2)}$$

Rule

For an if-else structure, consider the branch with the largest asymptotic complexity.

Rule 7: Logarithmic Loop

```
for(i = 1; i <= n; i = i * 2)
{
    statement;
}
```

Values of i are:

$$1,\ 2,\ 4,\ 8,\ 16, \ldots, n$$

After k iterations:

$$2^k = n$$

Taking logarithm:

$$k = \log_2 n$$

Therefore:

$$\boxed{O(\log n)}$$

Example 1: Analyze the Algorithm

```
1 for(i = 0; i < n; i++)
2 {
3     printf("%d", i);
4 }
```

Example 1: Analyze the Algorithm

```
for(i = 0; i < n; i++)
{
    printf("%d", i);
}
```

The loop executes n times.

Therefore:

$$T(n) = cn$$

$$\boxed{T(n) = O(n)}$$

Example 2: Analyze the Algorithm

```
for(i = 0; i < n; i++)
{
    for(j = 0; j < n; j++)
    {
        printf("*");
    }
}
```

Example 2: Analyze the Algorithm

```
for(i = 0; i < n; i++)
{
    for(j = 0; j < n; j++)
    {
        printf("*");
    }
}
```

Number of executions:

$$n \times n = n^2$$

Therefore:

$$\boxed{T(n) = O(n^2)}$$

Example 3: Analyze the Algorithm

```
i = 1;

while(i < n)
{
    printf("%d", i);
    i = i * 2;
}
```

Values:

$$1,\ 2,\ 4,\ 8, \ldots, n$$

Example 3: Analyze the Algorithm

```
i = 1;

while(i < n)
{
    printf("%d", i);
    i = i * 2;
}
```

Values:

$$1,\ 2,\ 4,\ 8, \ldots, n$$

Therefore:

$$2^k = n$$

$$k = \log_2 n$$

$$\boxed{T(n) = O(\log n)}$$

Example 4: Analyze the Algorithm

```
for(i = 0; i < n; i++)
{
    statement1;
}

for(i = 0; i < n*n; i++)
{
    statement2;
}
```

First loop: $O(n)$

Second loop: $O(n^2)$

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

Therefore: $\boxed{O(n^2)}$

Asymptotic Analysis: Summary

Three Important Notations

$\boxed{O(g(n))}$	Upper Bound
$\boxed{\Omega(g(n))}$	Lower Bound
$\boxed{\Theta(g(n))}$	Tight Bound

- Ignore constant factors.
- Ignore lower-order terms.
- Focus on the dominant term.
- Analyze growth as $n \to \infty$.

Understanding Big-O: What Does $O(n^3)$ Mean?

Important Idea

Big-O describes an **upper bound** on the growth rate of an algorithm.

Suppose an algorithm has:

$$T(n) = 5n^3 + 10n^2 + 20n + 50$$

Therefore:

$$\boxed{T(n) = O(n^3)}$$

But note:

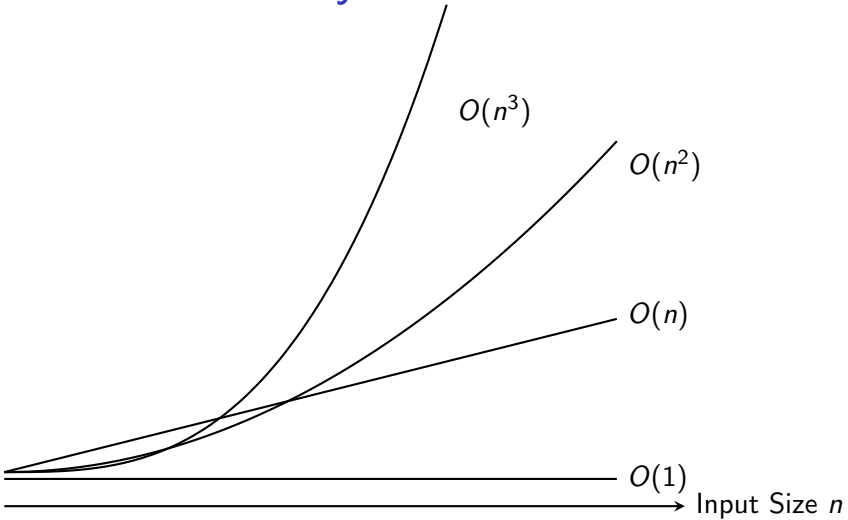
$$O(n^3) \text{ is not only } n^3$$

It represents a class of functions that grow **no faster than** a constant multiple of n^3.
Examples of functions that are also:

$$O(n^3):$$

$$\boxed{O(1), \quad O(\log n), \quad O(n), \quad O(n \log n), \quad O(n^2), \quad O(n^3)}$$

Big-O as a Growth Hierarchy



Understanding O, Ω and Θ

Consider:

$$f(n) = 3n^2 + 5n + 10$$

For large n, the dominant term is n^2.
Therefore:

$$\boxed{f(n) = \Theta(n^2)}$$

Since $\Theta(n^2)$ gives both an upper and lower bound:

$$f(n) = O(n^2)$$

and

$$f(n) = \Omega(n^2)$$

Key Takeaways

1. Algorithm analysis measures resource requirements.

Key Takeaways

1. Algorithm analysis measures resource requirements.
2. Time complexity measures computational work.

Key Takeaways

1. Algorithm analysis measures resource requirements.
2. Time complexity measures computational work.
3. Space complexity measures memory requirements.

Key Takeaways

1. Algorithm analysis measures resource requirements.
2. Time complexity measures computational work.
3. Space complexity measures memory requirements.
4. Best, average and worst cases describe different input scenarios.

Key Takeaways

1. Algorithm analysis measures resource requirements.
2. Time complexity measures computational work.
3. Space complexity measures memory requirements.
4. Best, average and worst cases describe different input scenarios.
5. Asymptotic notation describes growth for large n.

Key Takeaways

1. Algorithm analysis measures resource requirements.
2. Time complexity measures computational work.
3. Space complexity measures memory requirements.
4. Best, average and worst cases describe different input scenarios.
5. Asymptotic notation describes growth for large n.
6. O gives an upper bound.

Key Takeaways

1. Algorithm analysis measures resource requirements.
2. Time complexity measures computational work.
3. Space complexity measures memory requirements.
4. Best, average and worst cases describe different input scenarios.
5. Asymptotic notation describes growth for large n.
6. O gives an upper bound.
7. Ω gives a lower bound.

Key Takeaways

1. Algorithm analysis measures resource requirements.
2. Time complexity measures computational work.
3. Space complexity measures memory requirements.
4. Best, average and worst cases describe different input scenarios.
5. Asymptotic notation describes growth for large n.
6. O gives an upper bound.
7. Ω gives a lower bound.
8. Θ gives a tight bound.

Remember

Good algorithm design means controlling the growth of computational resources as input size increases.

Thank You

Questions?