Notes: Analysis of Algorithms

What is Analysis of Algorithms?

Analyzing how much time and memory an algorithm needs as the input size (n) increases.

Why Do We Analyze Algorithms?

- To compare performance of algorithms
- To check how well they scale with large inputs
- To identify inefficiencies and optimize programs

Key Concepts - Time Complexity

Describes how the running time grows with input size n.

Written in Big O Notation.

Common Big O Examples

O(1) - Constant time (e.g., Accessing arr[0])

O(n) - Linear time (e.g., Loop through array)

O(n^2) - Quadratic time (e.g., Nested loops, Bubble Sort)

O(log n) - Logarithmic time (e.g., Binary Search)

O(n log n)- Log-linear time (e.g., Merge Sort, Quick Sort avg case)

Key Concepts - Space Complexity

Measures how much extra memory the algorithm uses.

Cases in Time Complexity

Best Case - Fastest scenario (e.g., First match in Linear Search)

Average Case - Expected performance (e.g., Practical estimation)

Worst Case - Slowest scenario (standard) (e.g., No match in search)

Asymptotic Analysis

Notes: Analysis of Algorithms

Describes the growth rate of time/space usage as n → ∞.

Helps compare algorithms regardless of hardware.

Example: Algorithm Analysis

Code:

```
def sum_array(arr):
    total = 0
    for num in arr:
        total += num
    return total
```

Analysis:

- total = 0 → O(1)
- Loop runs n times → O(n)
- Combined: O(n)
- No extra space → O(1)

Types of Algorithm Analysis

Empirical - Run code and measure actual performance

Theoretical - Use Big-O math to predict behavior

Amortized - Spread cost of expensive ops over time

Summary

- Use Big O to describe growth, not exact time
- Consider both time and space
- Worst-case analysis is most common
- Focus on performance with input size (n)