Time Complexity in Data Structures and Algorithms

Prepared by Abdo

July 25, 2025

Contents

Chapter 1

Introduction

Time complexity is a measure that gives an idea of the amount of time an algorithm takes to run as a function of the length of the input. Understanding time complexity is crucial for evaluating the efficiency of algorithms.

1.1 What is Time Complexity?

Time complexity expresses the relationship between the size of the input and the number of operations required. It is commonly represented using Big-O notation.

Common Big-O Notations

Notation	Name	Example
$O(1)$	Constant Time	Accessing an array element
$O(\log n)$	Logarithmic Time	Binary Search
$O(n)$	Linear Time	Traversing an array
$O(n \log n)$	Linearithmic Time	Merge Sort, Quick Sort (avg)
$O(n^2)$	Quadratic Time	Bubble Sort, Insertion Sort
$O(2^n)$	Exponential Time	Recursive Fibonacci
$O(n!)$	Factorial Time	Solving TSP with brute force

Chapter 2

Visualizing Time Complexities

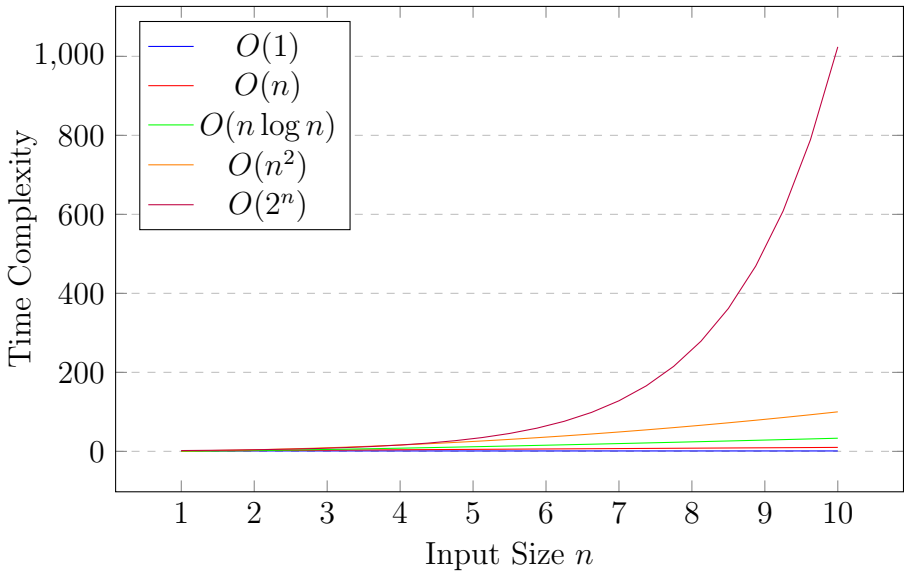

Figure 2.1: Graph comparing time complexities

Chapter 3

Best, Worst, and Average Cases

- **Best Case:** Minimum time taken for any input size.
- **Worst Case:** Maximum time taken for any input size.
- **Average Case:** Average time over all possible inputs.

Chapter 4

Time Complexity of Common Algorithms

4.1 Searching Algorithms

- Linear Search: $O(n)$
- Binary Search: $O(\log n)$

4.2 Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

4.3 Recursion Example – Fibonacci

```
int fibonacci(int n) {
    if(n <= 1)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Time Complexity: $O(2^n)$

Chapter 5

Space Complexity

Space complexity measures the total amount of memory space that the algorithm needs to run completely. It includes:

- Input space
- Auxiliary space

5.1 Examples

- Constant Space: $O(1)$ – using few variables
- Linear Space: $O(n)$ – storing array, recursion stack

Chapter 6

Tips to Improve Efficiency

- Choose efficient data structures (e.g., hash map over array)
- Avoid unnecessary loops
- Use memoization for recursion
- Break down complex problems

Chapter 7

Conclusion

Understanding time complexity helps you write better, faster, and more scalable programs. It plays a vital role in competitive programming, interviews, and real-world software development.