Complexity Analysis

Introduction: The Heart of Computer Science

In the vast world of computer science, where problems range from sorting numbers to training artificial intelligence, efficiency is a key concern. Every programmer, from beginners to industry experts, strives to write code that not only solves problems correctly but also does so efficiently. But how do we measure efficiency? How do we compare two algorithms that perform the same task? This is where complexity analysis comes into play.

Imagine you are tasked with searching for a name in a massive phone directory. You could scan every name sequentially or directly jump to the section where the name is likely to be found. The first method is simple but slow, while the second is much faster. This analogy captures the essence of algorithmic efficiency—the difference between brute force and optimized solutions.

In this guide, we will explore the fundamental concepts of time complexity and space complexity, breaking them down into their components, real-world implications, and mathematical representations. By the end, you will have a deep understanding of how to analyse algorithms effectively.

Asymptote

An asymptote in complexity analysis refers to the behaviour of an algorithm as the input size n approaches infinity. It describes the growth rate of the function that represents time or space complexity while ignoring constants and lower-order terms.

Asymptotic analysis helps determine how an algorithm scales by focusing on its dominant term, which has the highest impact as '*n*' grows large. For example, if an algorithm has a time complexity of:

$$T(n)=3n^2 + 5n + 7$$

The asymptotic complexity is $O(n^2)$ because as n à ∞, the term $3n^2$ dominates, and lower-order terms become negligible.

Types of Asymptotes

1. Big-O (O): Upper bound (worst case)
2. Omega (Ω): Lower bound (best case)
3. Theta (Θ): Tight bound (average case)

Asymptote Notations

What Do They Represent?

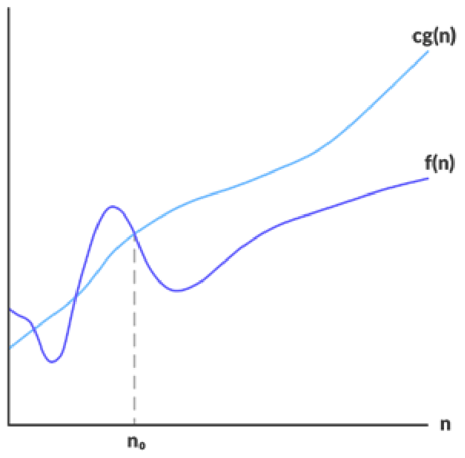
In complexity analysis, we use three primary notations to describe the growth of an algorithm's running time:

- Big O (O) → Upper Bound (Worst-Case)
- Big Omega (Ω) → Lower Bound (Best-Case)
- Big Theta (Θ) → Tight Bound (Average-Case)

These notations help us understand how an algorithm scales as the input size grows.

Big O Notation (O) – The upper bound

Big O notation describes the worst-case performance of an algorithm. It tells us the maximum amount of time an algorithm could take for a given input size 'n'. This helps in estimating the worst possible runtime in practical scenarios.



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

Mathematical Definition

An algorithm is said to be O(f(n)) if there exist constants c and n_0 such that:

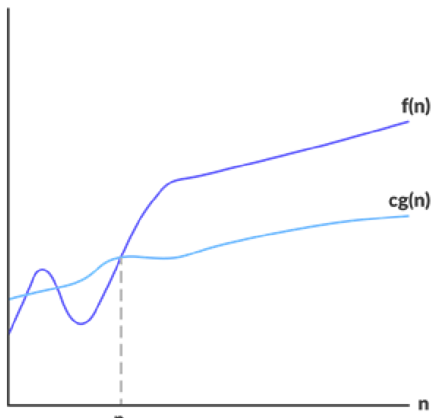
$$T(n) \leq c \cdot g(n) \qquad \text{for all } n \geq no$$

Where:

- T(n) is the actual time complexity function – f(n)
- g(n) is an asymptotic upper bound.
- c and no are constants to ensure the inequality holds.

Big Omega (Ω) – The Lower Bound

Big Omega notation describes the best-case performance of an algorithm. It provides a lower limit on the running time, meaning the algorithm at least takes this much time.



f(n) = Ω(g(n))

Mathematical Definition

An algorithm is said to be Ω(f(n)) if there exist positive constants c and n_0 such that:

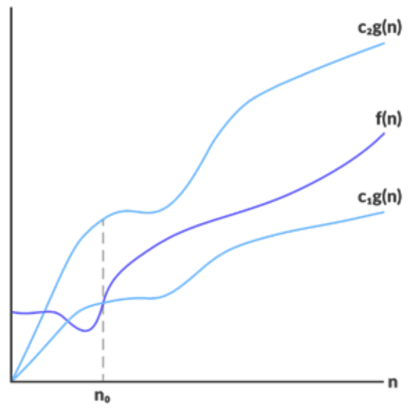
$$T(n) \geq c \cdot g(n) \qquad \text{for all } n \geq no$$

Where:

- T(n) is the actual time complexity function – f(n)
- g(n) is an asymptotic upper bound.
- c and no are constants to ensure the inequality holds.

Big Theta (Θ) – The Tight Bound

Big Theta notation defines a tight bound, meaning the algorithm takes at least Ω(f(n)) time and at most O(f(n)) time. It represents an average-case scenario in many cases.



f(n) = Θ(g(n))

Mathematical Definition

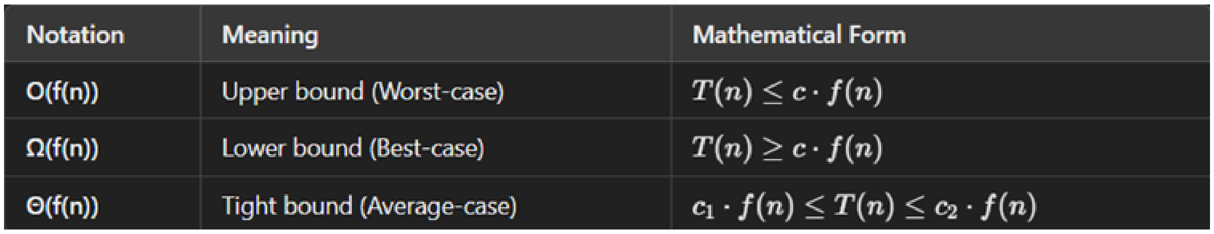
An algorithm is Θ(f(n)) if there exist positive constants c_1, c_2, and n_0 such that:

$$c1 \cdot g(n) \leq T(n) \leq c2 \cdot g(n) \qquad \text{for all } n \geq no$$

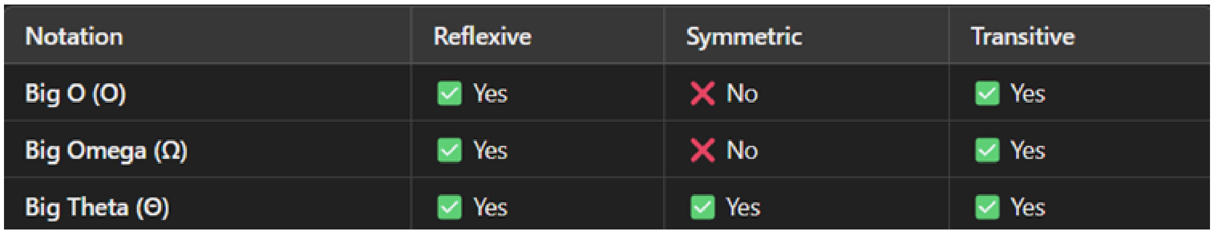
Where:

- T(n) is the actual time complexity function – f(n)
- g(n) is an asymptotic upper bound.
- c_1, c_2, and n_0 are constants to ensure the inequality holds.

Summary Table



Notation	Meaning	Mathematical Form
O(f(n))	Upper bound (Worst-case)	$T(n) \leq c \cdot f(n)$
Ω(f(n))	Lower bound (Best-case)	$T(n) \geq c \cdot f(n)$
Θ(f(n))	Tight bound (Average-case)	$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$



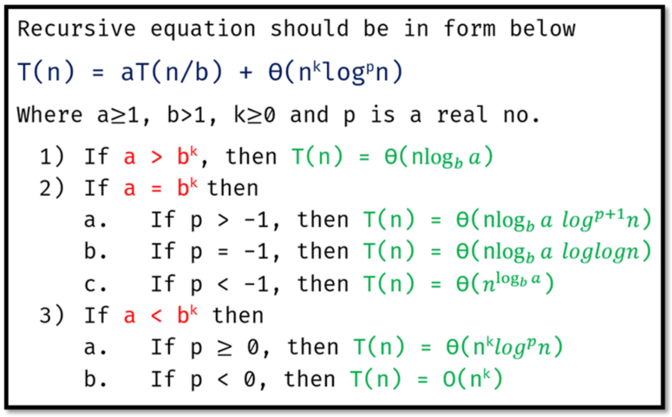
Notation	Reflexive	Symmetric	Transitive
Big O (O)	✅ Yes	❌ No	✅ Yes
Big Omega (Ω)	✅ Yes	❌ No	✅ Yes
Big Theta (Θ)	✅ Yes	✅ Yes	✅ Yes

Graphical Order (Fastest to Slowest):

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

The Master Theorem: Solving Recurrence Relations

For divide-and-conquer algorithms, the time complexity is often defined using recurrence relations:



Recursive equation should be in form below

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

Where $a \geq 1$, $b > 1$, $k \geq 0$ and p is a real no.

1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$
2) If $a = b^k$ then
   a. If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c. If $p < -1$, then $T(n) = \Theta(n^{\log_b a})$
3) If $a < b^k$ then
   a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

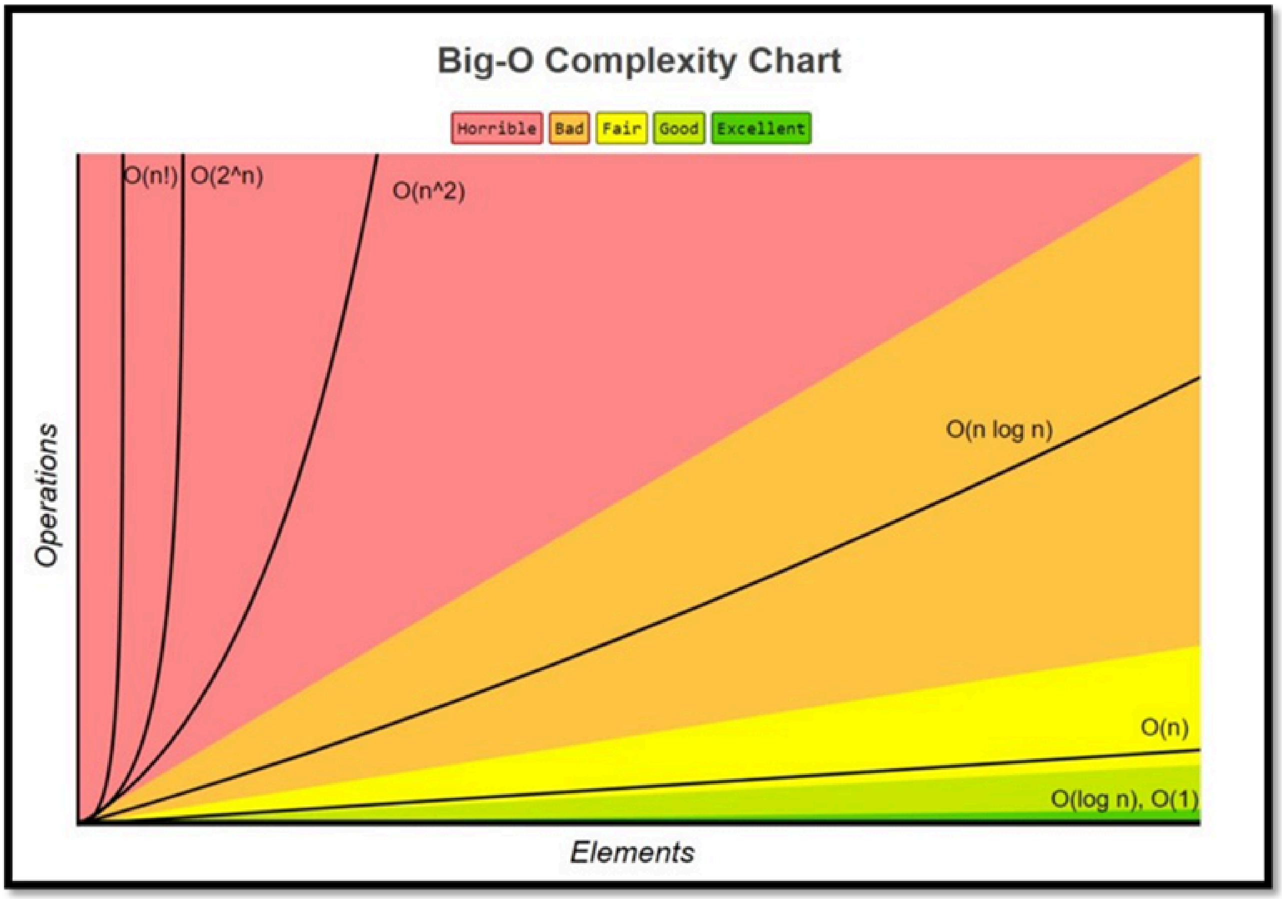
$$T(n) = aT(n/b) + f(n)$$

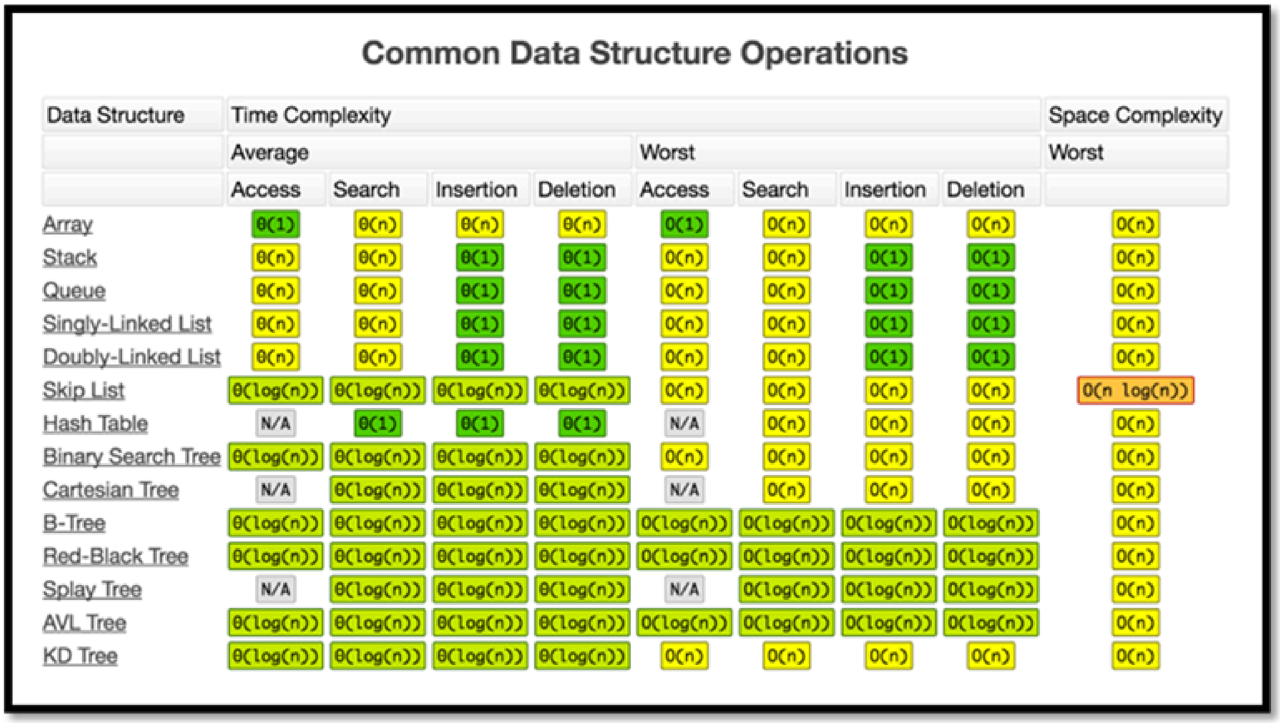
where:

- a = number of subproblems [Condition: a ≥ 1]
- b = factor by which input size decreases [Condition: b > 1]
- f(n) = time taken for combining solutions

CodeVerse

Google Developer Groups
On Campus • RV University

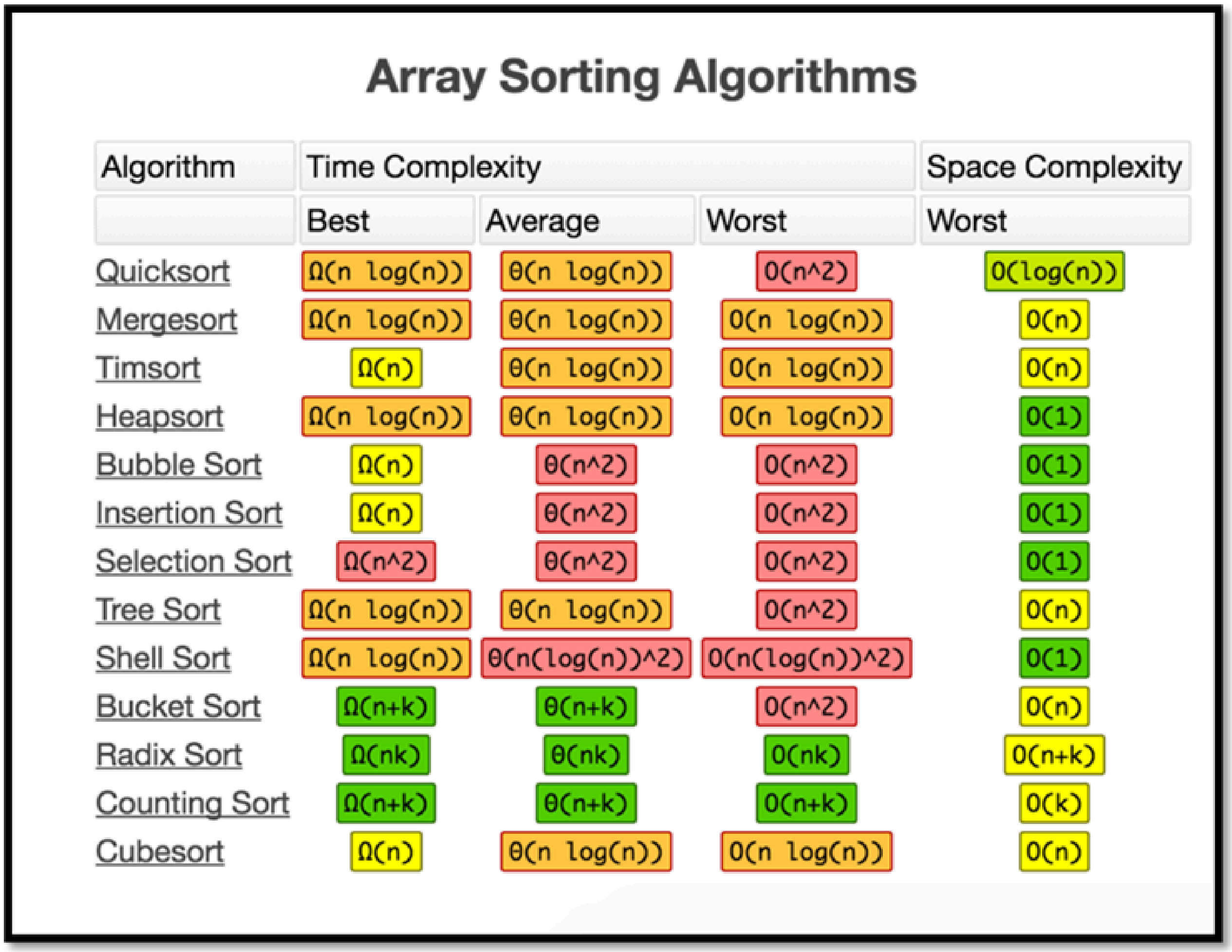




Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Google Developer Groups
On Campus • RV University



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Reference

[1] Programiz

[2] Properties of Asymptote - GFG

[3] Master theorem

[4] YouTube for Complexity Analysis - Shoeb Sir [RVU faculty] * **Recommended**

[5] Introduction to Algorithms - Thomas R Cormen

Questions:

1) Determine the time complexity of the following code snippet:

```
for i in range(n):
    for j in range(i, n):
        # constant time operation
```

(A) O(n)
(B) O(n log n)
(C) O(n^2)
(D) O(n^3)

2) Which of the following functions is asymptotically equivalent to $f(n) = 3n^2 + 5n + 7$?

(A) O(n)
(B) O(n^2)
(C) O(n^3)
(D) O(log n)

3) Consider two algorithms with time complexities O(n^2) and O(2^n) respectively. For large values of n, which algorithm is more efficient?

(A) O(n^2)
(B) O(2^n)
(C) Both have the same efficiency
(D) Cannot be determined

4) If an algorithm has a time complexity of O(log n), which of the following input sizes will result in the fewest number of operations?

(A) n = 16
(B) n = 64
(C) n = 128
(D) n = 256

5) Which of the following statements is true regarding Big-O notation?

(A) It provides the exact execution time of an algorithm.
(B) It describes the worst-case scenario of an algorithm's time complexity.
(C) It describes the best-case scenario of an algorithm's time complexity.
(D) It is only applicable to recursive algorithms.