ChatGPT

Time Complexity in Algorithms: Detailed Notes for Engineering Students

---

1. What is Time Complexity?

Time complexity is a way to represent the amount of time an algorithm takes to complete, as a function of the size of its input (usually denoted as **n**). It helps in comparing the efficiency of algorithms.

2. Why Time Complexity Matters

- To evaluate performance.
- Helps in selecting optimal algorithms.
- Important for scaling large applications.

3. Types of Time Complexities

Here are common time complexities in increasing order of growth:

Time Complexity	Name	Example Algorithm
O(1)	Constant Time	Accessing an element in an array
O(log n)	Logarithmic Time	Binary Search
O(n)	Linear Time	Traversing an array
O(n log n)	Linearithmic Time	Merge Sort, Heap Sort
O(n^2)	Quadratic Time	Bubble Sort, Insertion Sort
O(n^3)	Cubic Time	Matrix multiplication (naive)
O(2^n)	Exponential Time	Solving Tower of Hanoi
O(n!)	Factorial Time	Solving Traveling Salesman Brute-force

---

4. Best, Worst, and Average Case

- **Best Case**: Minimum time taken on any input of size n.
- **Worst Case**: Maximum time taken on any input of size n.
- **Average Case**: Expected time over all inputs of size n.

Example (Linear Search):

- Best: O(1) (first element)
- Worst: O(n) (last or not found)
- Average: O(n/2) ≈ O(n)

---

5. Big O, Omega Ω, and Theta Θ Notations

- **Big O (O)**: Upper bound of time (worst-case scenario)
- **Omega (Ω)**: Lower bound (best-case scenario)
- **Theta (Θ)**: Tight bound (average-case)

Example: If an algorithm runs in 3n + 2 steps:

- O(n), Ω(n), and Θ(n) — because all describe linear growth

---

6. How to Calculate Time Complexity

Steps:

1. Count the number of basic operations.
2. Express it in terms of input size **n**.
3. Eliminate constants and lower-order terms.

Example:

```
for i in range(n):
    print(i)
```

- This loop runs **n** times → Time Complexity = **O(n)**

---

7. Time Complexity of Common Algorithms

Algorithm	Best	Average	Worst
Binary Search	O(1)	O(log n)	O(log n)
Linear Search	O(1)	O(n)	O(n)
Bubble Sort	O(n)	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)
Insertion Sort	O(n)	O(n^2)	O(n^2)

---

8. Nested Loops Example

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

Each loop runs **n** times → Total operations = n × n = **O(n^2)**

---

9. Recurrence Relations in Time Complexity

Used to express time complexity of recursive algorithms.

Example:

```
def binarySearch(arr, left, right, x):
    if right >= left:
        mid = (left + right) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binarySearch(arr, left, mid-1, x)
        else:
            return binarySearch(arr, mid+1, right, x)
    else:
        return -1
```

Recurrence: T(n) = T(n/2) + c → **O(log n)**

---

10. Tips to Reduce Time Complexity

- Use efficient data structures (HashMap, Heap, etc.)
- Avoid nested loops where possible
- Use divide and conquer
- Use memoization and dynamic programming

---

11. Visual Representation of Growth Rates

```
O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(2^n) < O(n!)
```

---

12. Summary Table

Complexity	Description
O(1)	Constant time
O(log n)	Logarithmic (binary cuts)
O(n)	Linear
O(n log n)	Log-linear (divide & conquer)

Complexity	Description
O(n^2)	Quadratic (nested loops)
O(2^n)	Exponential (recursive growth)
O(n!)	Factorial (permutations)

---

13. Practice Problems

1. Calculate time complexity of:

```
for i in range(n):
    for j in range(i):
        print(i, j)
```

1. Time complexity of recursive Fibonacci?
2. Compare Merge Sort vs Bubble Sort in terms of time.

These notes are essential for understanding algorithm efficiency and preparing for technical interviews or academic exams.