Search 2

Name: Mariam magdy mostafa

What is Complexity?

In Simple Terms:

Complexity in computer science refers to how much time and memory (or other resources) an algorithm needs to solve a problem, especially as the size of the input increases.

It helps us evaluate and compare algorithms to see which ones are more efficient or scalable.

Why Is It Important?

When writing code, especially for large datasets or time-sensitive applications, we want to:

- Run programs **quickly**.
- Use **less memory**.
- Handle **large inputs** smoothly.
- Avoid performance issues

➢ Types of Complexity

1. Time Complexity

Definition:

Time complexity describes how the runtime of an algorithm grows **as the input size increases (denoted in Big-O notation).**

Measured Using: Big O Notation

Big O gives the upper bound **of growth, focusing on the** worst-case scenario**.**

Common Time Complexities:

Time Complexity	Name	Example
O(1)	**Constant Time**	**Accessing an array element arr[5]**
O(log n)	**Logarithmic Time**	**Binary Search**
O(n)	**Linear Time**	**Linear Search**
O(n log n)	**Linearithmic Time**	**Merge Sort, Quick Sort (average case)**
$O(n^2)$	**Quadratic Time**	**Nested loops (e.g., Bubble Sort)**
$O(2^n)$	**Exponential Time**	**Recursive algorithms like the naive Fibonacci**
O(n!)	**Factorial Time**	**Brute-force solutions to traveling salesman**

Example: Linear vs Binary Search

Linear Search (O(n)): Checks each element one by one.

Binary Search (O(log n)): Repeatedly halves the search space (works only on sorted data).

2. Space Complexity

Definition:

- **Measures memory usage relative to input size**

- **Also depends on input size and what data structures or variables are created.**

Examples:

O(1) – Constant space (e.g., swapping two variables).

O(n) – Linear space (e.g., storing an array of size n).

O(n^2) – Quadratic space (e.g., a 2D matrix).

How Do We Measure It?

We use asymptotic notation to express growth rates:

Big O (O-notation):

- **Describes the upper bound or worst-case scenario.**
- **Helps us understand how the algorithm behaves as input gets very large.**

Other Notations (less common):

- **Ω (Omega): Best-case performance.**
- **Θ (Theta): Average-case or tight bound (both upper and lower).**
- **P vs NP – A major unsolved problem asking whether problems solvable in polynomial time (P) can also be verified in polynomial time (NP).**

Why Complexity Matters?

- **Efficient algorithms save time, memory, and computational power.**
- **Especially important for large data sets, real-time systems, and embedded devices.**
- **Algorithm Selection: Helps choose the best algorithm for large datasets.**

  Example: Google Search uses highly optimized O(log n) algorithms.

- **System Scalability: Predicts how software behaves as input grows.**
- **Problem Tractability:**
- **Tractable = Solvable in polynomial time (P).**

- **Intractable = Requires exponential time (NP-Hard).**

What is Big Big-O notation?

Big-O notation is a mathematical way to describe the upper bound of an algorithm's time or space complexity. It tells us how an algorithm's runtime or memory usage grows as the input size (n) approaches infinity.

Why is it Called "Big O"?

- **The "O" stands for Order of growth.**
- **When we say something is O(n), we mean "the algorithm grows on the order of n."**

Big O Focuses On:

Aspect	Meaning
Time	How long it takes to run
Space	How much memory it uses
Input size	Usually represented by n
Growth rate	How fast time/space usage increases

Big-O vs Other Notations

Notation	Meaning	Example
Big-O (O)	Upper bound (worst-case).	"Takes at most n^2 steps."
Big-Ω (Ω)	Lower bound (best-case).	"Takes at least n steps."
Big-Θ (Θ)	Tight bound (exact growth).	"Takes exactly n log n steps."

Example:

- **Binary Search:**
  - **Best case:** Ω(1) **(element is middle).**
  - **Worst case:** O(log n)**.**
  - **Average case: Θ(log n).**

Big-O, Big-Ω (Omega), Big-Θ (Theta) – Best, Worst & Average Case Complexity Explained:

In computer science, we use asymptotic notations to describe how algorithms perform as input size grows. The three most important ones are:

1. **Big-O (O) → Worst-case complexity (Upper Bound)**
2. **Big-Ω (Ω) → Best-case complexity (Lower Bound)**
3. **Big-Θ (Θ) → Average-case complexity (Tight Bound)**

1. Big-O (O) – Worst-Case Complexity

Definition:

- **Describes the maximum time/space an algorithm can take.**
- **Represents the upper bound (algorithm will never perform worse** than this).

Example:

- **Linear Search (finding an element in an unsorted array):**
  - **Worst case: Element is at the end → O(n) (checks all n elements).**

Key Idea:

- **Helps ensure that even in** the **worst scenario**, the algorithm remains efficient enough.

2. Big-Ω (Ω) – Best-Case Complexity

Definition:

- **Describes the minimum time/space an algorithm can take.**
- **Represents the lower bound (algorithm will never perform better than this).**

Example:

- **Linear Search**:
  - **Best case: Element is at the first position → Ω(1) (finds it immediately).**

Key Idea:

- **Shows the best possible scenario**, but rarely used alone (since worst-case is more critical).

3. Big-Θ (Θ) – Average-Case Complexity

Definition:

- **Describes the expected time/space under random inputs.**
- **Represents a tight bound (algorithm usually performs within this range).**

Example:

- **Linear Search**:
  - **Average case: Element is somewhere in the middle → Θ(n/2) → Simplified to Θ(n)**.

Key Idea:

- **Most realistic for real-world performance analysis.**

Types of Search

1. Linear Search (Sequential Search)

How it Works:

- **Checks each element one by one until the target is found.**
- **Works on any list (sorted or unsorted).**

Complexity:

- **Time:**
  - **Best case: O(1) (element is first).**
  - **Worst case: O(n) (element is last or missing).**
  - **Average case: O(n).**

- **Space: O(1) (no extra memory needed).**

Use Case:

- **Small or unsorted datasets.**

2. Binary Search (Divide & Conquer)

How it Works:

- **Requires a sorted list.**
- Repeatedly divides the list in half and checks the middle element.

Complexity:

- **Time: O(log n) (halves search space each step).**
- **Space:**
  - **Iterative: O(1).**
  - **Recursive: O(log n) (call stack).**

Use Case:

- **Large, sorted datasets (e.g., dictionaries, phone books).**

3. Hashing (Hash Table Search)

How it Works:

- **Uses a hash function to map keys to indices in a table.**
- **Enables near-instant lookups (average case).**

Complexity:

- **Time:**
  - **Best/Average: O(1).**
  - **Worst: O(n) (if collisions occur).**

- **Space: O(n) (stores all elements).**

Use Case:

- **Databases, caches, dictionaries (e.g., Python `dict`).**

4. Breadth-First Search (BFS)

How it Works:

- **Explores a graph/tree** level by level **(neighbors first before deeper nodes).**
- **Uses a** queue **(FIFO) to track nodes to visit next.**

Complexity:

- **Time: O(V + E)** (V = vertices, E = edges).
- **Space: O(V) (stores all nodes in the queue).**

Use Case:

- **Shortest path in unweighted graphs.**
- **Web crawling, social network analysis.**