What is Time Complexity?

Time complexity measures how efficient an algorithm is as the input size increases. It's not the same as the actual time taken to run a program.

`Time Complexity != Execution Time`

Linear vs Binary Search

Linear Search

Best Case: Element at 1st index → 1 operation

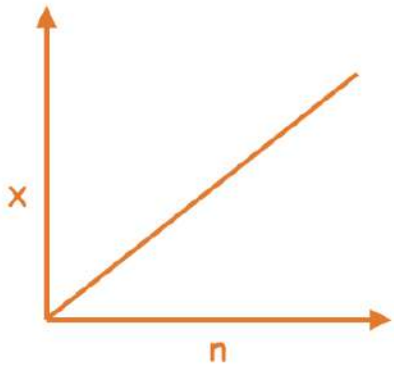
Average Case: Element at n/2 index → n/2 operations

Worst Case: Element not found → n operations

Time Complexity: O(n)

Requirement: Can work on unsorted arrays

```
n = 10 (Input Size)  -> 10 times loop run(x)
n = 100 (Input Size) -> 100 times loop run(x)
n = 1000 (Input Size) -> 1000 times loop run(x)
```



Binary Search

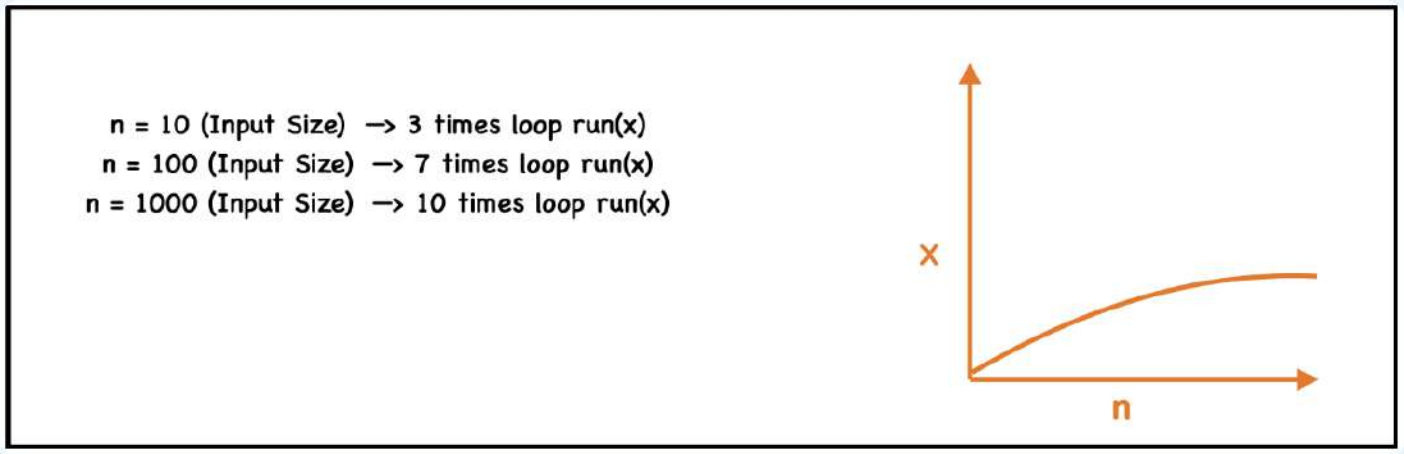
Best Case: Middle element matched → 1 operation

Average Case: $\log_2(n)$ operations

Worst Case: $\log_2(n)$ operations

Time Complexity: O(log n)

Requirement: Only works on sorted arrays



When we use `Linear Search` for an input size of 100, it runs 100 times, whereas `Binary Search` takes only 7 steps. This shows that Binary Search is more efficient. As the input size (n) increases, the way an algorithm behaves helps us understand how efficient it is. Also, the graph helps us understand that Binary Search is more efficient.

Big O Notation

It is nothing; just a symbol used to represent the worst-case complexity.

Code Examples of Time Complexity

`O(1)`

```
// Accessing 5th index element
int value = arr[5];
```

The time complexity is O(1) because we directly access the 5th index without any iteration.

`O(n)`

```
for(int i = 0; i < n; i++) {
    // do something
}
```

`O(log n)`

```
// e.g., Binary Search
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while(low <= high) {
        int mid = (low + high) / 2;
        if(arr[mid] == key) return mid;
        else if(arr[mid] < key) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

`O(n^2)` – Nested Loop

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        // do something
    }
}
```

`O(n log n)`

```
for(int i = 0; i < n; i++) {
    int temp = n;
    while(temp > 1) {
        temp = temp / 2;
        // do something
    }
}
```

`O(n^3)` – Triple Nested Loops

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < n; k++) {
            // do something
        }
    }
}
```

`O(2^n)`

```
// Recursive Fibonacci
int fib(int n) {
    if(n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

`O(n!)`

```cpp
// Permutation generator
void permute(string s, int l, int r) {
    if(l == r) {
        cout << s << endl;
    } else {
        for(int i = l; i <= r; i++) {
            swap(s[l], s[i]);
            permute(s, l + 1, r);
            swap(s[l], s[i]); // backtrack
        }
    }
}
```

Time Complexity Priorities

`O(1)` – Constant time

`O(log n)` – e.g., Binary Search

`O(n)` – e.g., Linear Search

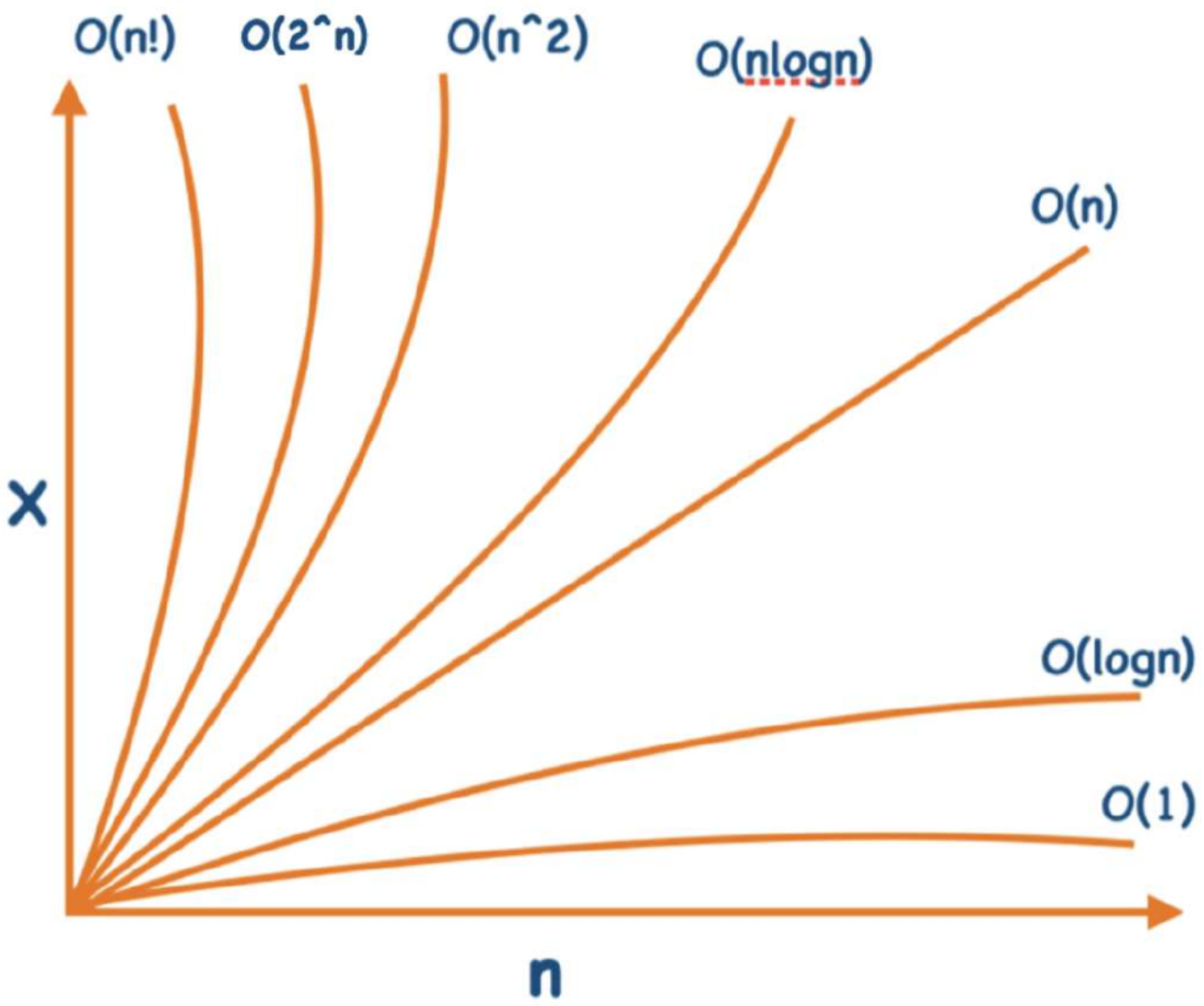
`O(n log n)` – e.g., Merge Sort

`O(n^2)` – e.g., Nested Loops

`O(n^3)` – e.g., Triple Nested Loops

`O(2^n)` – Recursion (e.g., Fibonacci)

`O(n!)` – e.g., Brute-force permutations



What is Space Complexity?

Space complexity refers to how much extra memory an algorithm uses.

Examples:

Access 5th element: `O(1)`

Find max with variable: `O(1)`

New array: `O(n)`

2D Matrix: `O(n^2)`