Big O Notation: Mathematical and Practical Explanation

1 Introduction to Big O Notation

Big O notation is a mathematical tool used in computer science to describe an algorithm's performance, focusing on how its runtime or space requirements grow with input size. It provides an asymptotic upper bound, emphasizing the worst-case scenario for large inputs.

2 Mathematical Definition

For functions $f(n)$ and $g(n)$, where n is the input size, we say:

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

if there exist positive constants c and n_0 such that for all $n \geq n_0$,

$$0 \leq f(n) \leq c \cdot g(n).$$

Here, $f(n)$ is the algorithm's actual complexity, and $g(n)$ is an upper-bound function. This implies $f(n)$ grows no faster than $g(n)$ scaled by c, for large n.

Common complexity classes include:

- $O(1)$: Constant time (independent of input size).
- $O(\log n)$: Logarithmic time (e.g., binary search).
- $O(n)$: Linear time (e.g., linear scan).
- $O(n \log n)$: Linearithmic time (e.g., merge sort).
- $O(n^2)$: Quadratic time (e.g., bubble sort).
- $O(2^n)$: Exponential time (e.g., recursive brute force).
- $O(n!)$: Factorial time (e.g., permutations).

Big O focuses on the dominant term. For example, if $f(n) = 3n^2 + 2n + 5$, then $f(n) = O(n^2)$.

3 Practical Examples

Below are examples in Python and C, with their time complexities analyzed.

3.1 Constant Time: $O(1)$

Accessing an array element by index is constant time, as it requires a fixed number of operations regardless of array size.

Python Example:

```
def get_first_element(arr):
    return arr[0]  # O(1) - direct access
```

C Example:

```
int get_first_element(int arr[], int size) {
    return arr[0];  // O(1) - direct access
}
```

3.2 Linear Time: $O(n)$

Finding the maximum in an unsorted list requires scanning each element, proportional to input size.

Python Example:

```
def find_max(arr):
    max_val = arr[0]
    for num in arr[1:]:  # Loops n-1 times
        if num > max_val:
            max_val = num
    return max_val
```

C Example:

```
int find_max(int arr[], int size) {
    int max_val = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    return max_val;
}
```

3.3 Logarithmic Time: $O(\log n)$

Binary search on a sorted array halves the search space each step, leading to logarithmic complexity.

Python Example:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
```

```
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

C Example:

```
int binary_search(int arr[], int size, int target) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

3.4 Quadratic Time: $O(n^2)$

Bubble sort uses nested loops, resulting in quadratic complexity due to pairwise comparisons.

Python Example:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

C Example:

```
void bubble_sort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

4 Comparison Table

The following table summarizes the complexities and their practical implications for an input size of $n = 100$.

Complexity	Example Algorithm	When to Use	Operations ($n = 100$)
$O(1)$	Array access	Instant results	~ 1
$O(\log n)$	Binary search	Sorted data, fast lookup	~ 7
$O(n)$	Linear search/max	Unsorted, simple scan	~ 100
$O(n^2)$	Bubble sort	Small datasets only	$\sim 5,000$

Table 1: Comparison of time complexities.

5 Conclusion

Big O notation helps evaluate algorithm efficiency. In practice, measure with tools like Python's `timeit` or C's `clock()` to validate theoretical complexities, as constants and hardware impact small inputs.