Table of Contents

What Is Big O Notation

Big O notation is a mathematical notation used in computer science to describe the upper bound or worst-case behavior of an algorithm or function. It represents the maximum growth rate of the algorithm's time complexity or space complexity as the input size approaches infinity.

- In Big O notation, we use the symbol "O" followed by a function to express the upper bound of the algorithm's complexity.
- The function typically represents the number of operations performed by the algorithm or the amount of space required.

Some Commonly Used Big O Notations:

1. O(1) – Constant Time:

The algorithm's running time or space requirements remain constant regardless of the input

size.

Example: Accessing an element in an array by index. It takes the same amount of time regardless of the size of the array.

2. O(log n) – Logarithmic Time:

The algorithm’s running time grows logarithmically with the input size.

Example: Binary search on a sorted array. At each step, the algorithm eliminates half of the remaining elements, reducing the search space logarithmically.

3. O(n) – Linear Time:

The algorithm’s running time increases linearly with the input size.

Example: Searching for an element in an unsorted array. In the worst case, the algorithm may need to traverse the entire array to find the element.

4. O(n log n) – Linearithmic Time:

The algorithm’s running time grows in a rate that is proportional to n multiplied by the logarithm of n.

Example: Merge sort. It divides the input array into smaller halves recursively and merges them in a sorted order. The time complexity grows in n log n as each division takes O(log n) time, and the merging step takes O(n) time.

5. $O(n^2)$ – Quadratic Time:

The algorithm's running time grows quadratically with the input size. It is commonly associated with nested loops or algorithms that involve comparing every element with every other element.

Example: Selection sort. It repeatedly finds the minimum element and swaps it with the current position. The algorithm requires nested loops, resulting in a quadratic time complexity.

6. $O(2^n)$ – Exponential Time:

The algorithm's running time grows exponentially with the input size.

Example: Generating all subsets of a set. As the size of the set grows, the number of subsets doubles, leading to an exponential increase in time complexity.

Analyze The Time Complexity Of The Algorithm, Using Big O Notation.

Example 1:

Let n represent the number of elements in the array.

C

```c
#include <stdio.h>

int find_max(int arr[], int length) {
    int max_value = arr[0];  // Assume the first element is the
maximum

    for (int i = 1; i < length; i++) {
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }

    return max_value;
}

int main() {
    int arr[] = {5, 8, 2, 10, 3};
    int length = sizeof(arr) / sizeof(arr[0]);

    int max = find_max(arr, length);
    printf("Maximum value: %d\n", max);

    return 0;
}
```

1. Initializing max_value with arr[0] takes constant time and can be considered O(1).
2. The for loop iterates through the array from index 1 to length – 1, where length is the length of the array. The loop runs length – 1 times.
3. Within the loop, the comparison if (arr[i] > max_value) and the subsequent assignment max_value = arr[i] both take constant time and can be considered O(1).
4. The return statement also takes constant time and can be considered O(1).

Thus, find_max's time complexity is:

- The initialization step takes O(1).
- The for loop runs length – 1 times, so it has a time complexity of O(length).
- The remaining constant-time operations also take O(1).

As a result, find_max's time complexity is O(length), where length is the array's length.

Example 2:

C

```c
#include <stdio.h>

int main() {
    int i;
    for (i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

The loop runs for a fixed number of iterations, specifically from 1 to 10. Since the loop does not depend on any variable or input size, the time complexity is constant.

Therefore, the time complexity of this code snippet is O(1).

Example 3:

C

```c
#include <stdio.h>

int main() {
    int i,n;
    printf("Enter a number");
    scanf("%d",&n);
    for (i = 1; i <= n; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

1. The scanf function for getting user input takes constant time and can be considered O(1).
2. The for loop iterates from 1 to n, where n represents the user input. The loop runs n times.
3. Inside the loop, the printf function prints the value of i. The printf function takes constant time as it performs a fixed number of operations. Thus, it can be considered O(1).

Therefore, the time complexity of the for loop is O(n) since the loop iterates n times.