Time Complexity

Asymptotic Notation

Asymptotic Notation in Data Structure is a way to describe how the time or space needed by an algorithm grows as the size of the input grows. It helps us understand how efficient an algorithm is when we have very large inputs.

Imagine you have different ways to solve a problem, like sorting a list of numbers. Some ways are fast when the list is short, but they might become very slow when the list is long. Asymptotic notation helps us compare these methods by focusing on how they perform when the input size gets really big.

In simple terms, asymptotic notation helps us predict how an algorithm will perform and make better choices when writing programs.

Types of Asymptotic Notation in Data Structure

1. Big O Notation (O)

Big O Notation is used to describe the upper bound of an algorithm's running time. It tells us the maximum time an algorithm could take to complete, given the size of the input.

How it Measures the Upper Bound?
Big O notation focuses on the worst-case scenario. It helps us understand the maximum time or space an algorithm will require, regardless of the input's specifics.

Common Examples:

- O(1): Constant time. The algorithm takes the same amount of time regardless of the input size. Example: Accessing an element in an array.
- O(n): Linear time. The algorithm's running time grows linearly with the input size. Example: Iterating through a list.
- O(n^2): Quadratic time. The running time grows quadratically as the input size increases. Example: Bubble sort.

2. Omega Notation (Ω)

Omega Notation is used to describe the lower bound of an algorithm's running time. It tells us the minimum time an algorithm will take to complete, given the size of the input.

How it Measures the Lower Bound?
Omega notation focuses on the best-case scenario. It helps us understand the least amount of time or space an algorithm will require.

Common Examples:

- Ω(1): Constant time. The algorithm takes at least a constant amount of time, regardless of the input size.
- Ω(n): Linear time. The algorithm takes at least linear time as the input size grows.

3. Theta Notation (Θ)

Theta Notation is used to describe the exact bound of an algorithm's running time. It tells us the average-case scenario, where the running time is bound both above and below by the same function.

How it Measures the Exact Bound?
Theta notation focuses on the typical running time of an algorithm. It shows that the algorithm's running time is both at least and at most a certain function of the input size.

Common Examples:

- Θ(1): Constant time. The algorithm takes a constant amount of time, regardless of the input size.
- Θ(n): Linear time. The algorithm's running time grows linearly with the input size.

Different Types of Time Complexities

1. Constant Time: O(1)

An **algorithm** is said to have a constant time complexity when the time to complete does not depend on the size of the input data. This is the ideal scenario for any operation.

Examples:

- Accessing any element in an array by index.

- Imagine you always place a returned book back in a designated spot at the front of the library regardless of how many books are there. The time it takes you to place this book doesn't depend on the number of books in the library—it's always the same, a constant time operation.

2. Logarithmic Time: O(log n)

Logarithmic time complexity occurs when the algorithm reduces the problem size by a factor with each step. Algorithms that operate this way become faster than linear time as the size of the input data increases.

Examples:

- Binary search is a classic example of logarithmic time complexity, where the dataset is halved with each iteration.
- Consider the process of looking for a specific book in a library where books are perfectly organized alphabetically. Instead of going through each book, you can start in the middle, and if the book you want is later in the alphabet, you skip the first half, or if it's earlier, you skip the second half. Each step cuts the number of books you need to check in half, significantly reducing the search time.

3. Linear Time: O(n)

An algorithm is said to have linear time complexity when the time to complete is directly proportional to the input size. This means if you double the input size, the runtime will also double.

Examples:

- Scanning an **array** or a **linked list** from start to finish
- Suppose you need to check if a book exists in the library, and the books are not in any particular order. You would potentially have to check each book until you find the one you're looking for or confirm it's not there. The time it takes to complete this task grows linearly with the number of books.

4. Linearithmic Time: O(n log n)

This complexity arises when an algorithm performs a logarithmic operation (like divide and conquer) multiple times linearly across the data set. Common sorting algorithms like mergesort and heapsort exhibit linearithmic time complexity.

They are highly efficient for large datasets and are preferred over simpler sorting methods like bubble sort.

5. Quadratic Time: $O(n^2)$

Algorithms with quadratic time complexity are generally characterized by two nested loops. Each element of the input data is processed multiple times, making these algorithms inefficient as the data size grows.

Examples:

- Simple sorting algorithms like bubble sort, insertion sort, and selection sort.
- Imagine you decide to organize all the books in the library by hand from scratch, comparing each book with every other book to decide the order. The amount of time and comparisons needed grows quadratically with the number of books. This is similar to the inefficient sorting methods where each item is compared to every other item.

6. Cubic Time: $O(n^3)$

Cubic time complexity appears in algorithms involving three nested loops. This is generally seen in more complex mathematical computations such as matrix multiplication or solving three-body problems in physics.

7. Exponential Time: O(2^n)

Algorithms with exponential time complexities double the amount of work with every additional element in the input data.

These are usually seen in brute-force solutions of complex problems, like calculating the Fibonacci sequence using a naive recursive approach or solving the traveling salesman problem through all permutations of cities.

8. Factorial Time: O(n!)

Factorial time complexity grows extremely fast with the increase in input size and is generally unsustainable even for relatively small n.

Space Complexity

This type of complexity is common in algorithms that generate all possible permutations of a dataset, such as solving the traveling salesman problem via a brute-force approach

Space complexity is a way to measure how much memory an algorithm needs to run. It tells us the total amount of space or memory an algorithm will use from start to finish. This includes the space needed for all the variables, data structures, and any extra space used during the execution.

Let’s define space complexity with a real-life example:

Algorithm space complexity is like the amount of space you need to store all your school supplies in your backpack. Imagine you have different subjects and each subject has its own set of books, notebooks, and stationery. The space complexity of your backpack is the total space needed to fit all these items.

If you have a simple subject like Math, you might only need one book and a notebook, taking up a small amount of space. But for a project-heavy subject like Art, you might need multiple sketchbooks, paints, brushes, and more, taking up much more space. Similarly, in algorithms, some tasks require only a little memory, while others need a lot more.

Understanding space complexity helps us choose the best way to pack our backpack (or write our algorithm) so that we use the least amount of space possible. This ensures we can fit everything we need without running out of room

Space complexity is composed of two main components: fixed part and variable part.

1. Fixed Part

This includes the space required for constants, simple variables, fixed-size data structures, and the code itself. It is the memory that does not change regardless of the input size.

Examples:

- Space needed for storing constants like numbers or fixed-size arrays.
- Memory used by the program code and instructions.

2. Variable Part

This includes the space required for dynamic memory allocation, recursive stack space, and temporary variables whose size depends on the input size.

Examples:

- Space needed for dynamic data structures like **linked lists**, **trees**, or **graphs**, which grow with the input size.
- Memory used for the recursion stack during recursive function calls.
- Temporary variables and data structures used during the execution of the **algorithm**.

3. Example Calculation

We are taking here the example of an iterative algorithm. The algorithm is to find the maximum element in an array.

```
def find_max(arr):
    max_val = arr[0]
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]
    return max_val
```

Fixed Part:

- Memory for variable max_val: O(1)
- Memory for loop counter i: O(1)

Variable Part:

- Memory for the input array arr: O(n)

Total Space Complexity:

`O(1)+O(n)=O(n)`

To learn more you can visit:

https://www.geeksforgeeks.org/time-complexity-and-space-complexity/

https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

https://www.w3schools.com/dsa/dsa_timecomplexity_theory.php