Asymptotic Notation

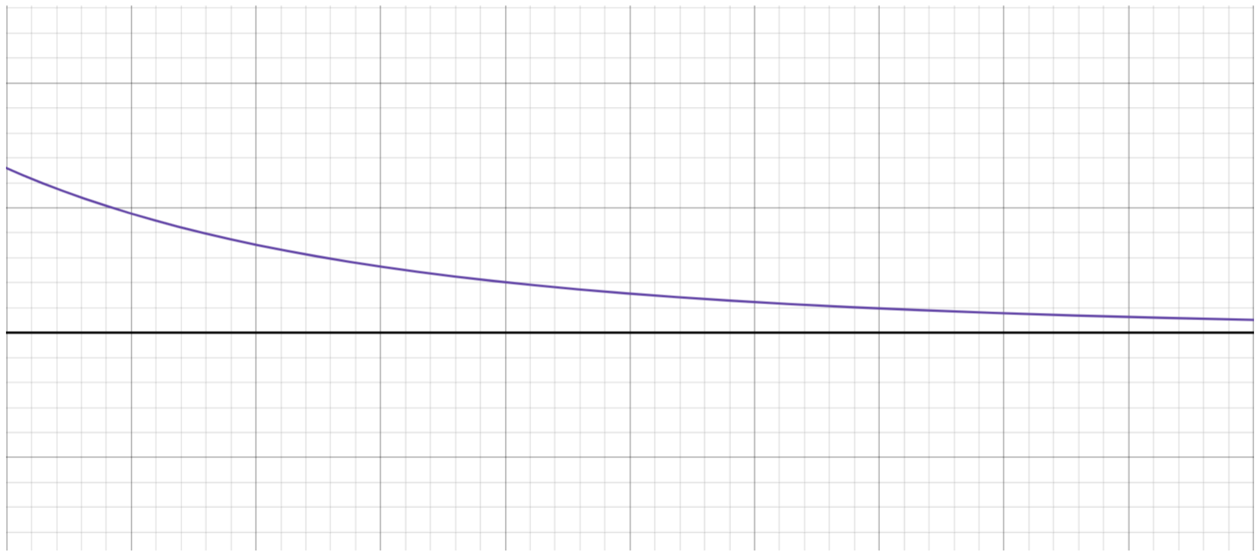
By: Aaryan Vinod Kumar

When analyzing an algorithm, we often want to know how much time or memory it will require. However, the **actual time** depends on many external factors like:

- The processor speed
- The programming language
- The system's load at runtime

Asymptotic notation removes these inconsistencies by focusing only on the *growth of the algorithm* as the input size, denoted by **n**, becomes large.

Asymptote: a straight line that continually approaches a given curve but does not meet it at any finite distance.



Here, the purple line is a function (say, f(x)) and the black line is its asymptote, which will come closer and closer to f(x) but never touch it.

In algorithm analysis, the curve is the actual performance of your algorithm, and the asymptote is a mathematical function that describes how the performance behaves as input size increases.

What is “n”?

- **n** is the size of the input.
- It could represent the number of elements in an array, the number of nodes in a graph, or the length of a string.
- As n increases, we observe how the algorithm’s performance scales.

Types of Asymptotic Notation

1. Big O Notation — O(f(n))

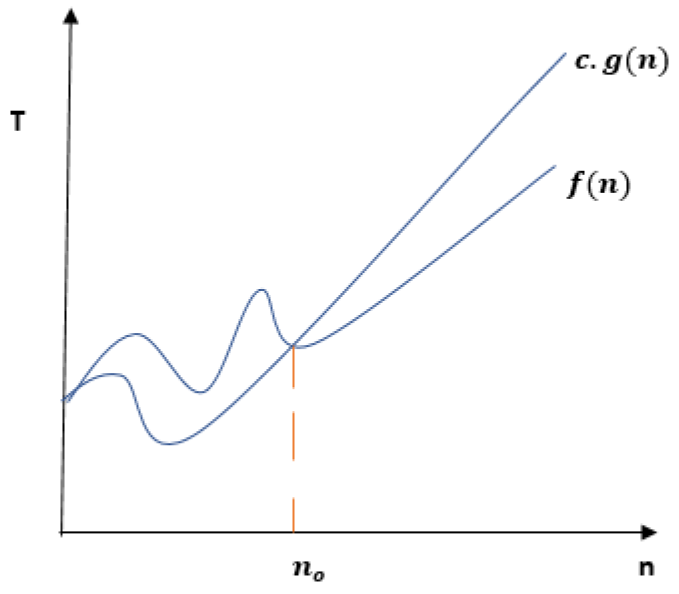
Purpose: Describes the upper bound or worst-case performance.

Big O provides a limit on the amount of time (or space) an algorithm will take in the worst possible scenario. It helps us know the maximum amount of resources the algorithm might consume.

Mathematically:
A function $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that:

$$f(n) \leq c \times g(n) \text{ for all } n \geq n_0$$

This means: beyond a certain input size (n_0), the function f(n) will never grow faster than some constant multiple of g(n).

- 

Examples:

- Linear search: O(n) — worst case when the target is not found.
- Binary search: O(log n) — each step halves the input.

cont.

2. Big Omega Notation — $\Omega(f(n))$

Purpose: Describes the lower bound or best-case performance.

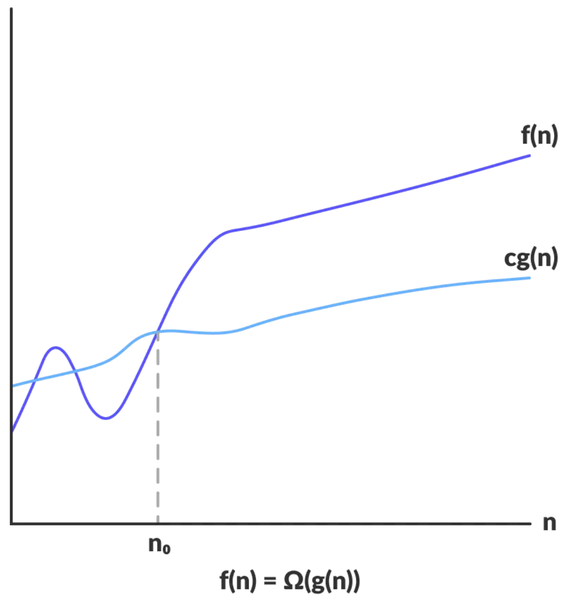
Big Omega tells us the least amount of time (or space) an algorithm will require under the best possible input conditions. It is not as commonly used in interviews but is important for complete analysis.

Mathematically:

$f(n) = \Omega(g(n))$ if there exist constants c and n_0 such that:

$$f(n) \geq c \times g(n) \text{ for all } n \geq n_0$$

It guarantees that the algorithm won't be *faster than this* in the best case.



Examples:

- Linear search: Ω(1) — best case when the target is the first element.
- Bubble sort (on sorted data): Ω(n) — one pass is enough to confirm order.

cont.

3. Big Theta Notation — Θ(f(n))

Purpose: Describes the tight bound — when both best and worst-case complexities are the same.

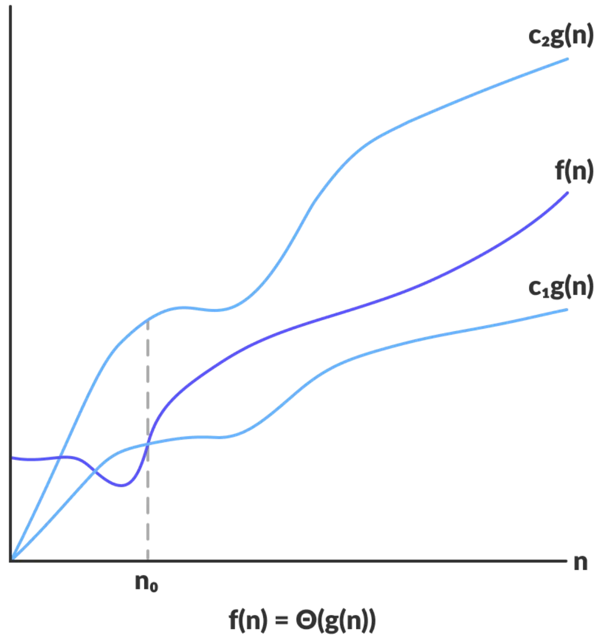
Theta notation provides an exact idea of how an algorithm behaves overall. It's used when the algorithm's time or space usage is consistently predictable.

Mathematically:

$f(n) = \Theta(g(n))$ if there exist constants c_1, c_2, and n_0 such that:

$$c_1 \times g(n) \le f(n) \le c_2 \times g(n) \text{ for all } n \ge n_0$$

This is a double-sided bound — meaning the algorithm grows neither faster nor slower than g(n) beyond a certain point.



Examples:

- Traversing all elements once: $\Theta(n)$
- A constant-time operation: $\Theta(1)$

cont.

Growth Rate Comparison

Notation	Name	Description	Example
$O(1)$	Constant	Does not change with input size	Accessing an array element
$O(\log n)$	Logarithmic	Input size is halved each step	Binary search
$O(n)$	Linear	Grows proportionally with input	Single loop over elements
$O(n \log n)$	Linearithmic	Slightly more than linear	Merge sort, heap sort
$O(n^2)$	Quadratic	Double loop over data	Bubble sort, selection sort
$O(2^n)$	Exponential	Doubles with each input increase	Solving subset problems
$O(n!)$	Factorial	Tries every possible arrangement	Brute-force permutations

Real-Life Analogy: Searching a Name

Imagine a phone book with one million names:

- **O(1)**: You magically know the page and go straight to the name.
- **O(log n)**: You keep dividing the book in half to find the right section.
- **O(n)**: You scan each name line by line.
- **O(n^2)**: You compare each name with every other name for sorting.

Misconceptions Clarified

1. **Big O is not actual time**
   It only shows how the resource usage increases as input grows.
2. **O(1) does not mean instant**
   It means the operation takes the same time regardless of input size, not that it's fast.
3. **Big O is not always enough**
   It shows the upper limit. For complete analysis, also consider Ω and Θ when appropriate.

Importance of Asymptotic Notation

- Used in analyzing and comparing algorithms.
- Essential in interviews and competitive programming.
- Helps build scalable and efficient software.

So that ends the brief explanation over Asymptotic notation. In the next document, we'll see how to apply these notations over real programs.

Image credits:

https://habr.com/en/articles/559518/

https://www.programiz.com/

Next page for last document's solution.

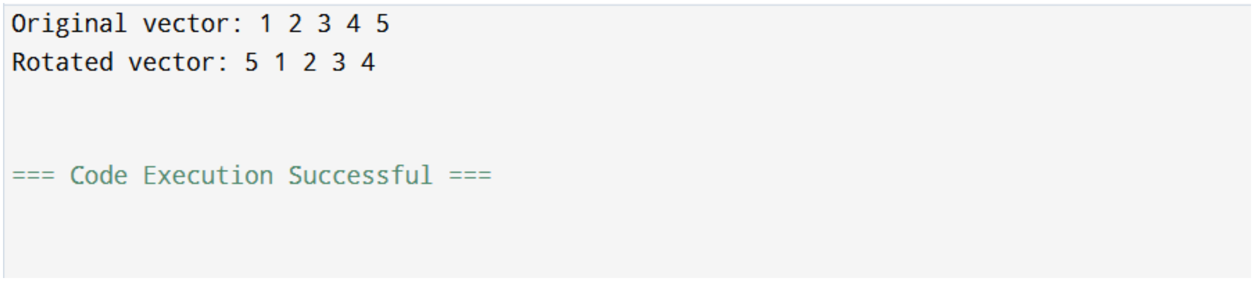
Solution to the question in previous document:

Q. **Write a C++ program to rotate a vector to the right by one position.** The last element should move to the front, and all others should shift one step to the right.

Program:

```cpp
#include <iostream>

#include <vector>



void rotateRightByOne(std::vector<int>& vec) {

    int n = vec.size();

    if (n == 0) return;



    int last = vec[n - 1];

    for (int i = n - 1; i > 0; --i) {

        vec[i] = vec[i - 1];

    }

    vec[0] = last;

}



int main() {

    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::cout << "Original vector: ";
    for (int num : vec) std::cout << num << " ";
    std::cout << "\n";

    rotateRightByOne(vec);

    std::cout << "Rotated vector: ";
    for (int num : vec) std::cout << num << " ";
    std::cout << "\n";

    return 0;
}
```

Output:



```
Original vector: 1 2 3 4 5
Rotated vector: 5 1 2 3 4


=== Code Execution Successful ===
```