The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. **The efficiency is measured with the help of asymptotic notations.**

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of **change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis**.

Asymptotic notations are the **mathematical notations** used to **describe the running time** of an algorithm when **the input tends towards a particular value or a limiting value.**

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

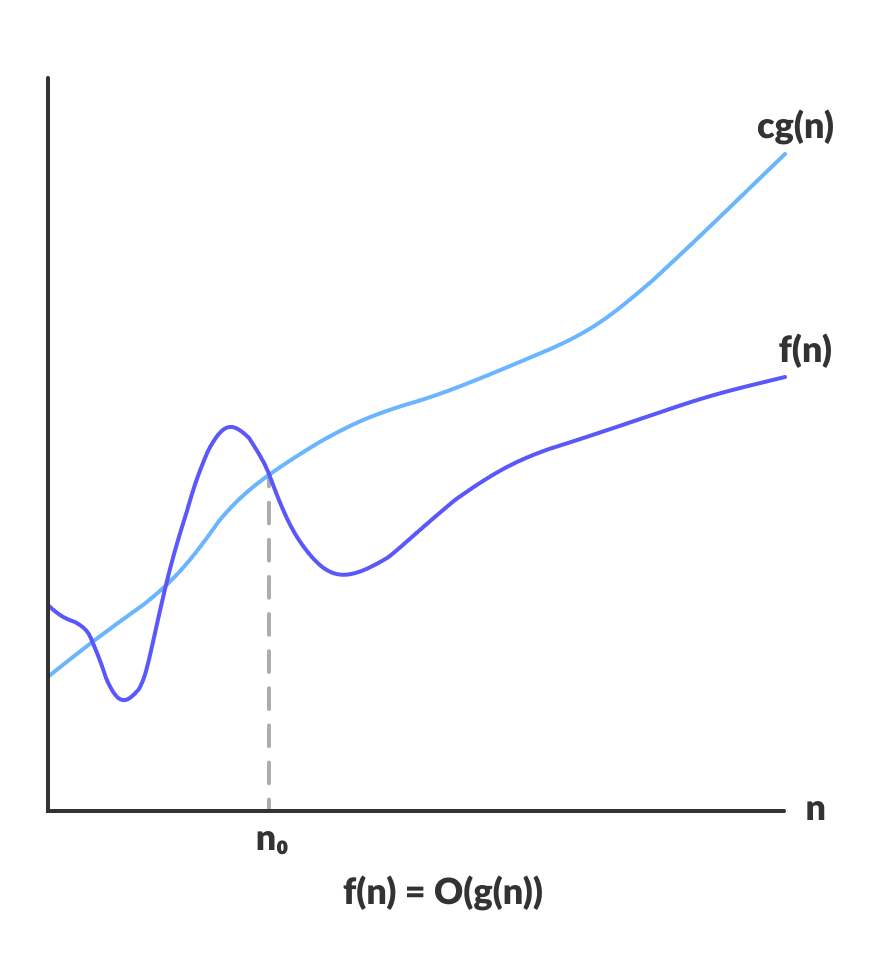
But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

* Big-O notation
* Omega notation
* Theta notation

**Big-O Notation (O-notation)**

* Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.
* 

O(g(n)) = {f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0}

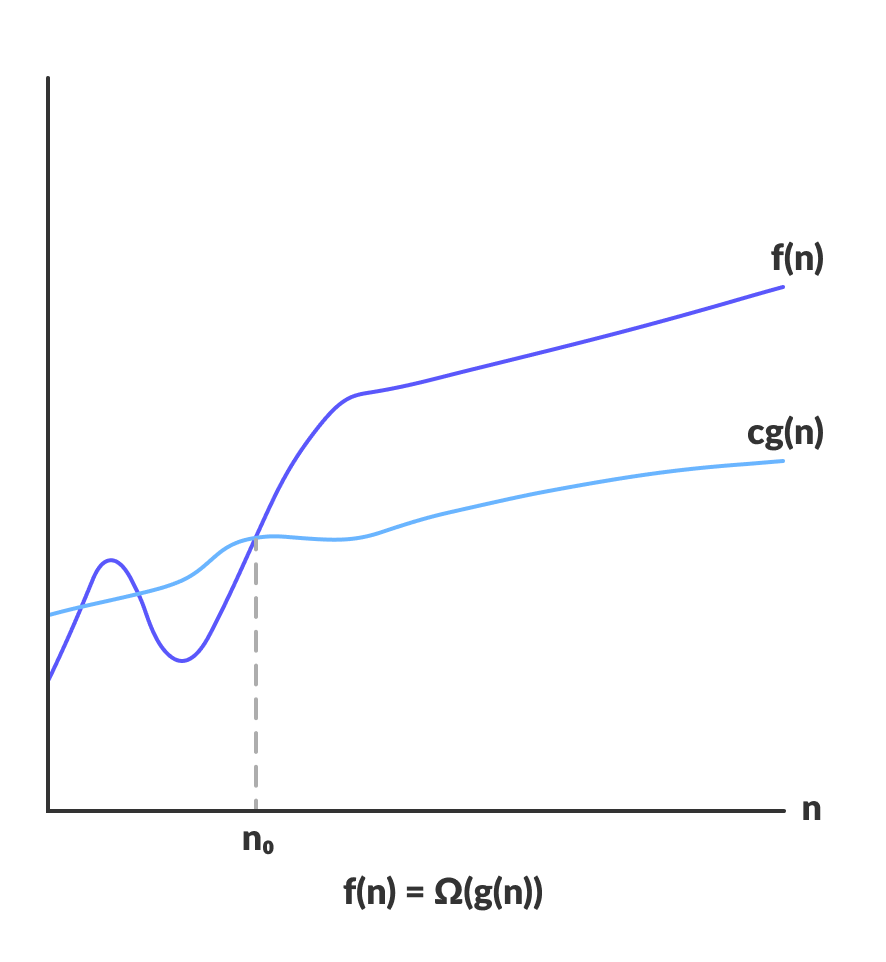
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyse an algorithm as we are always interested in the worst-case scenario.

**Omega Notation (Ω-notation)**

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Omega gives the lower bound of a function

Ω(g(n)) = { f(n): there exist positive constants c and n0

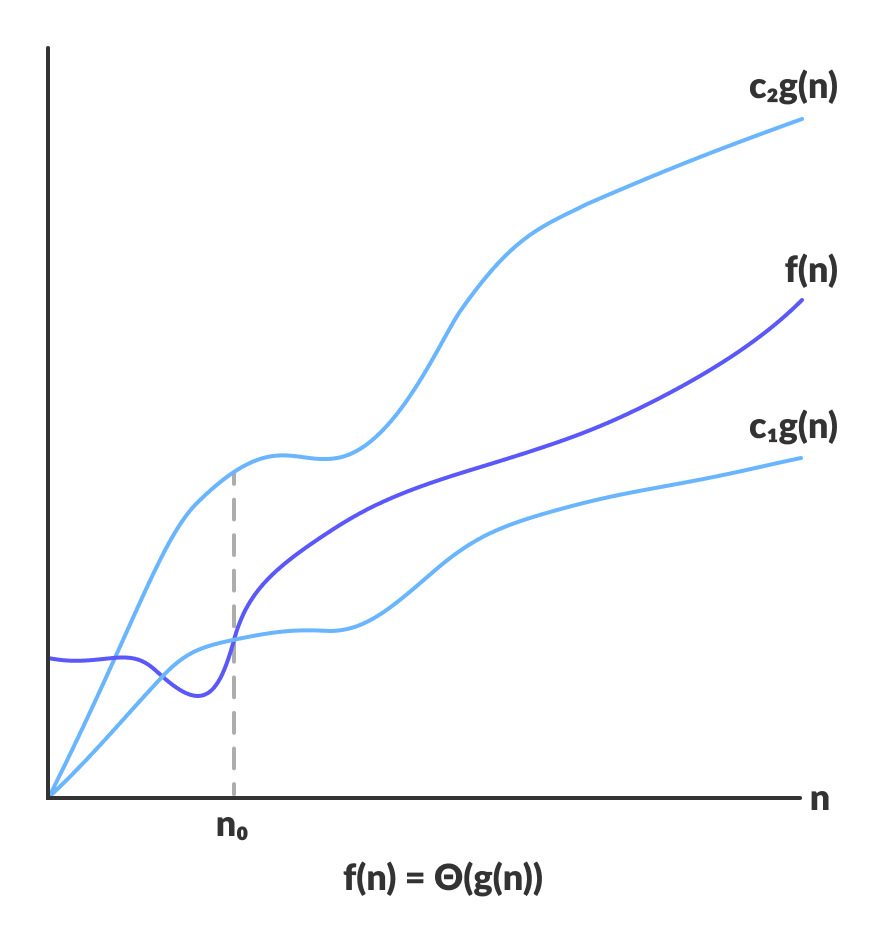
such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

**Theta Notation (Θ-notation)**

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Theta bounds the function within constants factors

For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

**SORTING ALGORITHMS**

BUBBLE SORT

**Bubble sort** is [a sorting algorithm](https://www.programiz.com/dsa/sorting-algorithm) that compares two adjacent elements and swaps them until they are not in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

## Bubble Sort Complexity

|  |  |
| --- | --- |
| **Time Complexity** |  |
| Best | O(n) |
| Worst | O(n2) |
| Average | O(n2) |
| **Space Complexity** | O(1) |
| **Stability** | Yes |

### Complexity in Detail

Bubble Sort compares the adjacent elements.

|  |  |
| --- | --- |
| Cycle | Number of Comparisons |
| 1st | (n-1) |
| 2nd | (n-2) |
| 3rd | (n-3) |
| ....... | ...... |
| last | 1 |

Hence, the number of comparisons is

(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2

nearly equals to n2

Hence, **Complexity:** O(n2)

Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is n\*n = n2

### 1. Time Complexities

* **Worst Case Complexity:** O(n2)  
  If we want to sort in ascending order and the array is in descending order then the worst case occurs.
* **Best Case Complexity:** O(n)  
  If the array is already sorted, then there is no need for sorting.
* **Average Case Complexity:**O(n2)  
  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

### 2. Space Complexity

* Space complexity is O(1) because an extra variable is used for swapping.
* In the **optimized bubble sort algorithm**, two extra variables are used. Hence, the space complexity will be O(2).