# Asymptotic notation

## Introduction

Asymptotic notation is a way of expressing the time or space complexity of an algorithm in terms of mathematical functions. The most commonly used asymptotic notations are Big O, Omega, and Theta.

1. **Big O Notation (O):** It represents the upper bound of an algorithm's running time. In simpler terms, it gives the worst-case time complexity of an algorithm.

Example: If an algorithm has a time complexity of O(n^2), it means the running time grows quadratically with the size of the input.

1. **Omega Notation (Ω):** This provides the lower bound of an algorithm's running time. It represents the best-case time complexity.

Example: If an algorithm has a time complexity of Ω(n), it means the running time grows at least linearly with the size of the input.

1. **Theta Notation (Θ):** This notation provides both upper and lower bounds, giving a tight bound on the algorithm's running time.

Example: If an algorithm has a time complexity of Θ(n), it means the running time grows linearly with the size of the input, and the upper and lower bounds match.

These notations are useful for analyzing the efficiency of algorithms without getting bogged down by hardware-specific details. They allow us to make general statements about an algorithm's performance as the input size grows.

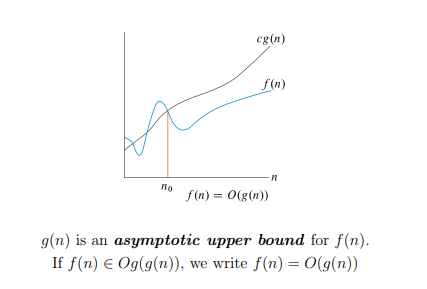
## Big O Notation

Big O notation is a mathematical notation used in computer science to describe the upper bound or worst-case performance of an algorithm. It provides a way to express how the running time or space requirements of an algorithm grow in relation to the size of the input.

Here are the key points about Big O notation:

1. **Definition:** Big O notation is defined as follows:

If a function g(n) is an upper bound on the running time of an algorithm for every input size n greater than a certain value n₀, then the algorithm's time complexity is O(g(n)).



Mathematically, we express this as:

In simpler terms, O(g(n)) represents the set of functions that grow no faster than g(n) for sufficiently large input sizes.

1. **Worst-Case Time Complexity:** Big O notation is often used to describe the worst-case scenario for an algorithm. It gives an upper limit on how the running time of the algorithm increases as the input size grows.
2. **Example:** Suppose we have an algorithm with a time complexity of O(n^2), where n is the size of the input. This means that the running time of the algorithm grows no faster than a quadratic function of the input size. In the worst-case scenario, the algorithm's performance is bounded by this quadratic function.
3. **Simplified Notation:** Big O notation provides an asymptotic upper bound, and it is used to simplify the analysis of algorithms. It focuses on the dominant term of the function, ignoring constant factors and lower-order terms. For example, O(2n^2 + 3n + 1) is simplified to O(n^2) because the quadratic term dominates for large values of n.
4. **Comparison with Omega Notation:** While Big O notation provides an upper bound on the running time (worst-case scenario), Omega notation provides a lower bound (best-case scenario). Together, they can give a more complete picture of an algorithm's performance.

In summary, Big O notation helps us understand the upper limits of an algorithm's efficiency, particularly in the worst-case scenario. It is widely used in algorithm analysis and provides a concise way to express how an algorithm's performance scales with input size.

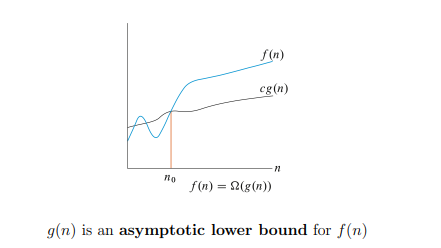
## Omega Notation

Omega notation (Ω) is a mathematical notation used in computer science to describe the lower bound or best-case performance of an algorithm. It provides information about the minimum growth rate of an algorithm's running time with respect to the size of the input.

Here are the key points about Omega notation:

1. **Definition:** Omega notation is defined as follows:

If a function g(n) is a lower bound on the running time of an algorithm for every input size n greater than a certain value n₀, then the algorithm's time complexity is Ω(g(n)).



Mathematically, we express this as:

In simpler terms, Ω(g(n)) represents the set of functions that grow at least as fast as g(n) for sufficiently large input sizes.

1. **Best-Case Time Complexity:** Omega notation is often used to describe the best-case scenario for an algorithm. It gives us information about the lower limit of the algorithm's efficiency, considering the most favorable conditions.
2. **Example:** Suppose we have an algorithm with a best-case time complexity of Ω(n), where n is the size of the input. This means that, in the best-case scenario, the algorithm's running time grows at least linearly with the size of the input. It won't perform better than linear time for any input.
3. **Comparison with Big O Notation:** While Big O notation provides an upper bound on the running time (worst-case scenario), Omega notation provides a lower bound (best-case scenario). Together, they can give a more complete picture of an algorithm's performance.

If an algorithm has a time complexity of Ω(g(n)) and O(f(n)), it means that the algorithm's running time is bounded both from below by g(n) and from above by f(n).

In summary, Omega notation helps us understand the lower limits of an algorithm's efficiency and is particularly useful for describing the best-case scenario. It complements Big O notation, which provides information about the upper limits and worst-case scenarios.

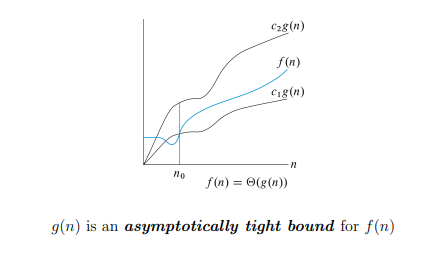
## Theta Notation

Theta notation (Θ) is another mathematical notation used in computer science to describe both the upper and lower bounds of an algorithm's running time. It provides a more precise characterization of the growth rate of an algorithm by specifying a tight range in which the running time lies.

Here are the key points about Big Theta notation:

1. **Definition:** Big Theta notation is defined as follows:

If a function g(n) is both an upper and lower bound on the running time of an algorithm for every input size n greater than a certain value n₀, then the algorithm's time complexity is Θ(g(n)).



Mathematically, we express this as:

In simpler terms, Θ(g(n)) represents the set of functions that grow at the same rate as g(n) for sufficiently large input sizes.

1. **Tight Bound:** The key feature of Big Theta notation is that it provides a tight or asymptotically exact bound on the running time of an algorithm. It means that the algorithm's performance is both upper and lower bounded by a specific function.
2. **Example:** If an algorithm has a time complexity of Θ(n), it means that the running time grows linearly with the size of the input, and the upper and lower bounds match. The algorithm's performance is within a constant factor of a linear function for sufficiently large input sizes.
3. **Comparison with Big O and Omega:** Big Theta notation is more specific than Big O and Omega notations. While Big O provides an upper bound (worst-case scenario) and Omega provides a lower bound (best-case scenario), Big Theta provides a tight bound that encapsulates both scenarios.

If an algorithm has a time complexity of Θ(g(n)), it implies that the worst-case and best-case performances are both proportional to g(n) within a constant factor.

In summary, Big Theta notation is used to describe the precise growth rate of an algorithm by providing both upper and lower bounds. It is particularly useful when we want to express the tightest possible relationship between an algorithm's running time and the size of its input.

## Why Big O?

Big O notation is often preferred and widely used in algorithm analysis for several practical reasons:

1. **Simplicity and Simplicity in Analysis:**
   * Big O notation provides a simple and high-level view of an algorithm's efficiency by focusing on the upper bound or worst-case scenario.
   * It allows for a straightforward comparison between algorithms and their scalability without getting into complex details.
2. **Abstraction from Constant Factors and Lower-Order Terms:**
   * Big O notation abstracts away constant factors and lower-order terms, providing a simplified representation that emphasizes the dominant factor affecting algorithmic complexity.
   * This abstraction makes it easier to compare the general efficiency of algorithms without being concerned about hardware specifics or exact implementation details.
3. **Ease of Communication:**
   * Big O notation facilitates communication about algorithmic complexity among researchers, developers, and educators.
   * It offers a common language to express the scalability of algorithms, making it easier for individuals with different backgrounds to understand and discuss computational efficiency.
4. **Worst-Case Scenario Focus:**
   * In many practical scenarios, understanding the worst-case time complexity is crucial for designing reliable systems. Big O notation inherently emphasizes the upper bound, providing insight into how an algorithm behaves in unfavorable conditions.
5. **Useful for Large Input Sizes:**
   * Big O notation is particularly useful when dealing with large input sizes, where the dominant term in the complexity function becomes the primary factor influencing performance.
   * It allows for quick assessments of how an algorithm scales as the input size grows without delving into intricate details.
6. **Tool for Algorithm Selection:**
   * When choosing an algorithm for a specific task, Big O notation helps developers make informed decisions about trade-offs between time and space complexity.
   * It guides the selection of algorithms that meet performance requirements for a given problem size.

While Big O notation is preferred for its simplicity and broad applicability, it's essential to recognize that it provides an upper bound and may not capture the full nuances of an algorithm's behavior. In some cases, considering Omega (Ω) or Theta (Θ) notations might be necessary for a more comprehensive understanding, especially when analyzing best-case scenarios or providing precise bounds.

## Exercises

1. Suppose that for inputs of size n on a particular computer, insertion sort runs in 8n 2 steps and merge sort runs in 64n log n steps. For which values of n does insertion sort beat merge sort?
2. What is the smallest value of n such that an algorithm whose running time is 100n 2 runs faster than an algorithm whose running time is 2n on the same machine?