Asymptotic notations are big O, big Theta (Θ), and big Omega (Ω).

Big O Notation (O)

Purpose: Describes the upper bound or worst-case scenario of an algorithm's time complexity.

Meaning: It represents the maximum time an algorithm will take to complete a task concerning the input size, in the worst-case scenario.

Example: Bubble sort algorithm has time complexity of O(n^2), where 'n' is the number of elements in the array. This is because Bubble Sort may require multiple passes (up to 'n' passes) through the array, and within each pass, it compares and potentially swaps each element.

Big Omega Notation (Ω):

Purpose: Describes the lower bound or best-case scenario of an algorithm's time complexity.

Meaning: It represents the minimum time an algorithm will take to complete a task concerning the input size, in the best-case scenario.

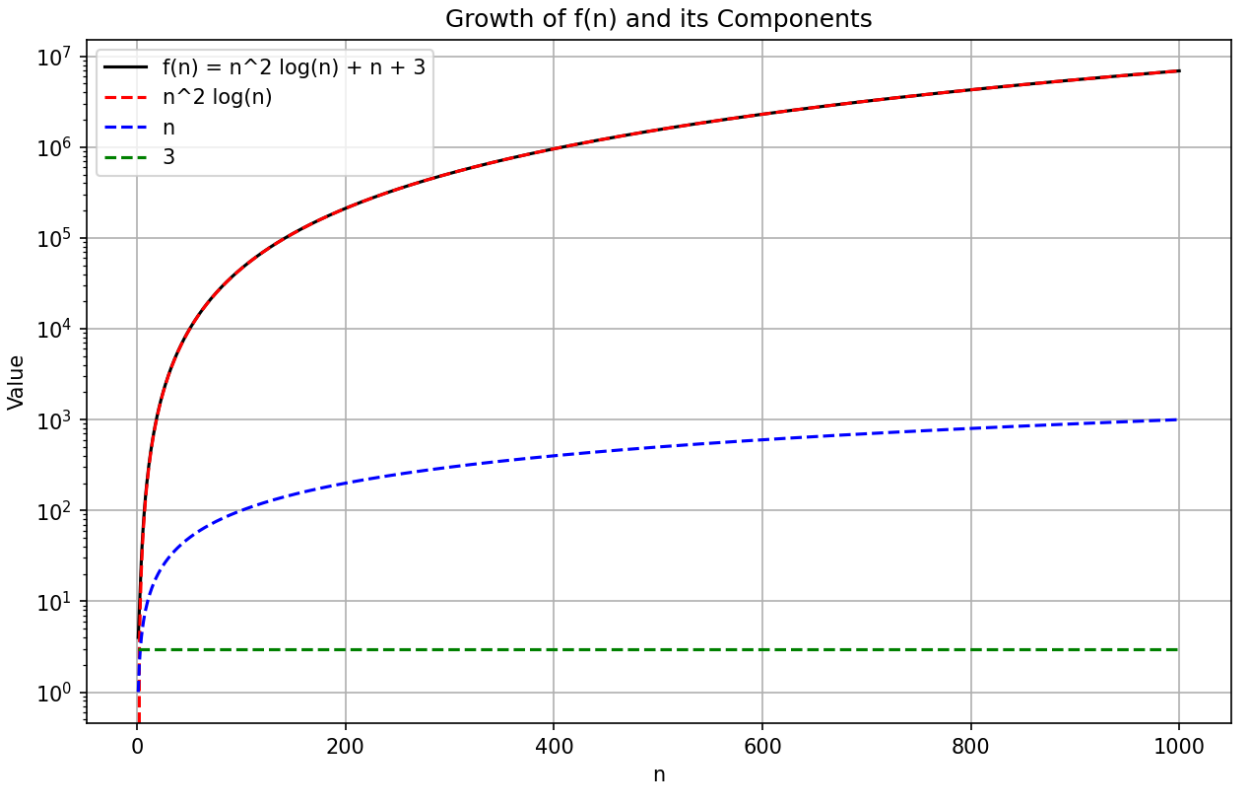
Example: Bubble sort algorithm has time complexity in its best-case scenario is Ω(n), where 'n' is the number of elements in the array. This is because it still needs to iterate through the array once to ensure its sorted but doesn't require any swaps.

Examples of Big O Notation (O):

1) f(n) = **(n^2)*(logn) + n + 3**
   Big O Notation **: O((n^2)*(logn))**

   Analysis :
   1) **n^2 log n** : This is a polynomial-logarithmic term, where the polynomial part (n^2) is the dominant factor, but it's also scaled by a logarithmic factor (log n)
   2) **n** : A linear term, which grows slower than the polynomial term.
   3) **3**: A constant term, which becomes negligible as n grows

   Among these, the term n^2 log n grows the fastest. The n and 3 terms contribute very little to the growth rate compared to for large values of Therefore, they are ignored in the Big O notation, which focuses on the upper bound of growth rate
   Graphical Representation

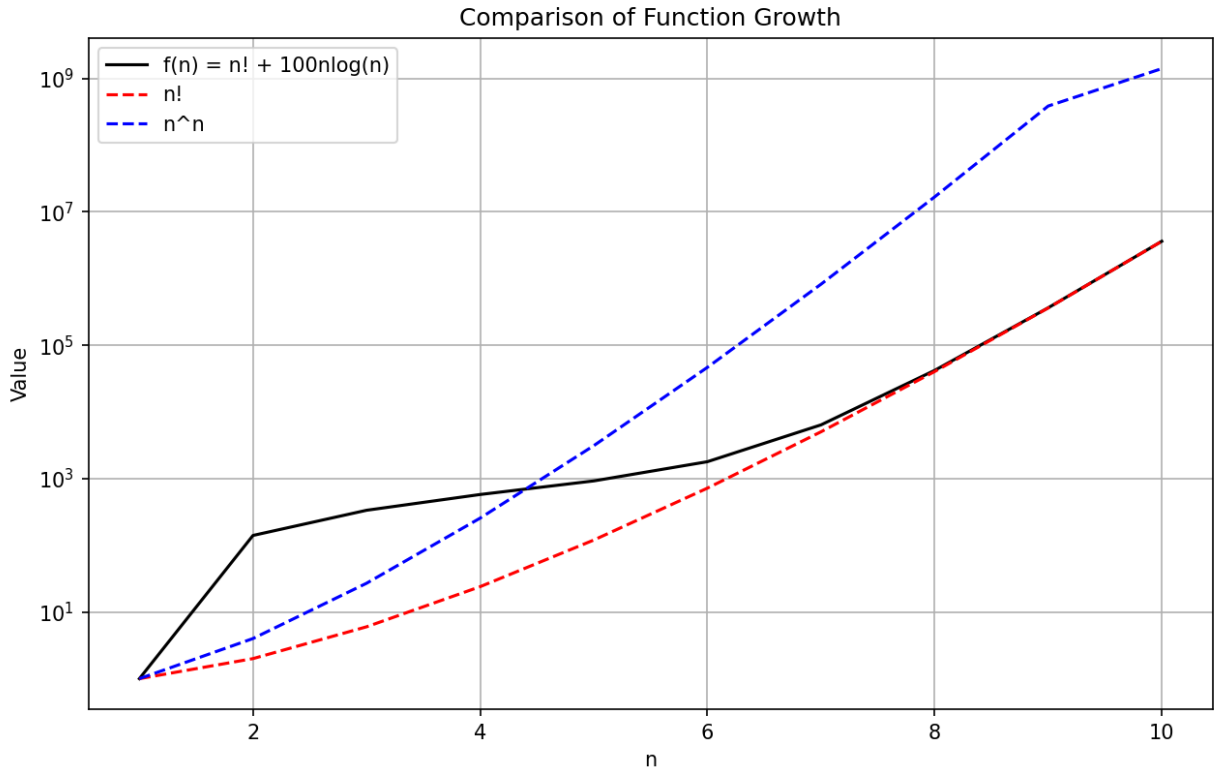


2) f(n) = **n! + 100nlogn**
Big O Notation: O(**n!**)

Analysis

1) n! is an extremely rapidly growing term, much faster than polynomial, exponential, or logarithmic growth rates.
2) 100nlogn: is a polynomial-logarithmic term, which grows significantly slower than a factorial term.

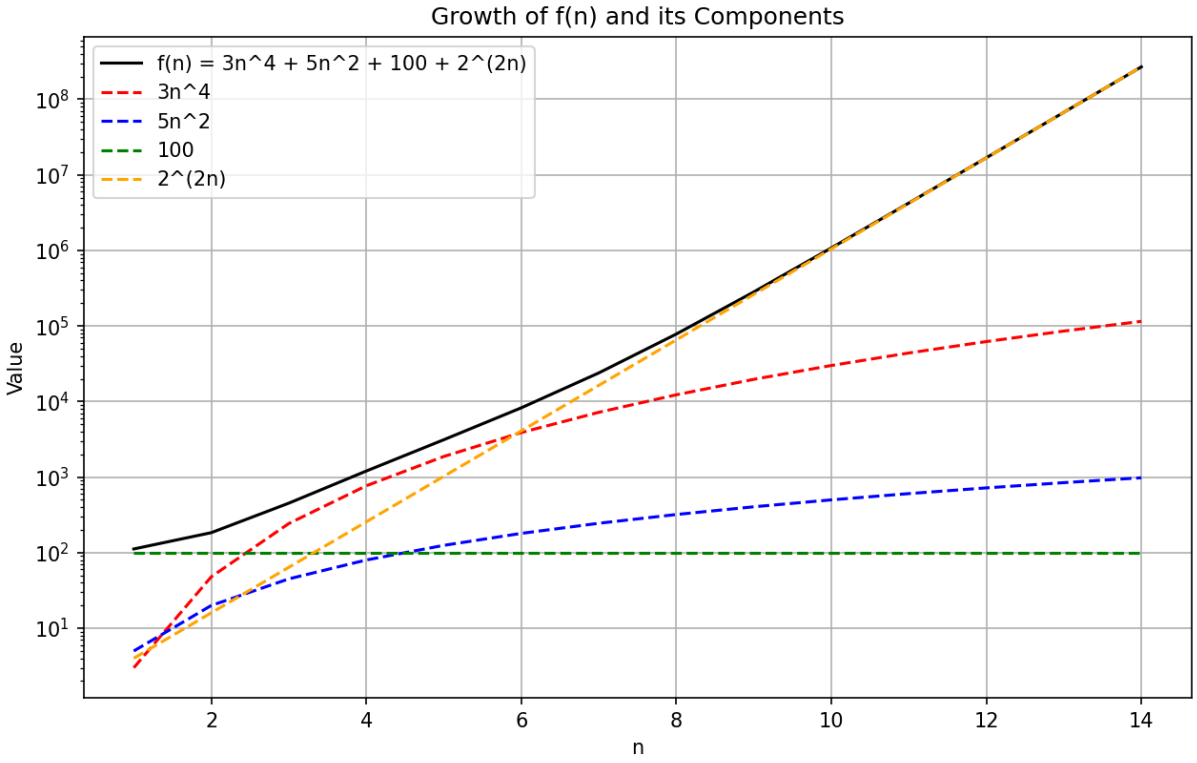
Since factorial growth n! is outpaces polynomial-logarithmic growth 100nlogn by a vast margin, especially as n becomes large.



3) f(n) = $\mathbf{3n^3 + 5n^2 + 100 + 2^{2n}}$
Big O Notation: $\mathbf{O(4^n)}$

Analysis
For small values of n, the polynomial terms might seem significant, but as n increases, the exponential term 4^n starts to dominate the growth of the function.
Even the highest-degree polynomial term $3n^3$ will be dwarfed by 4^n
The constant term 100 is insignificant in terms of growth rate and can be ignored in Big O notation.

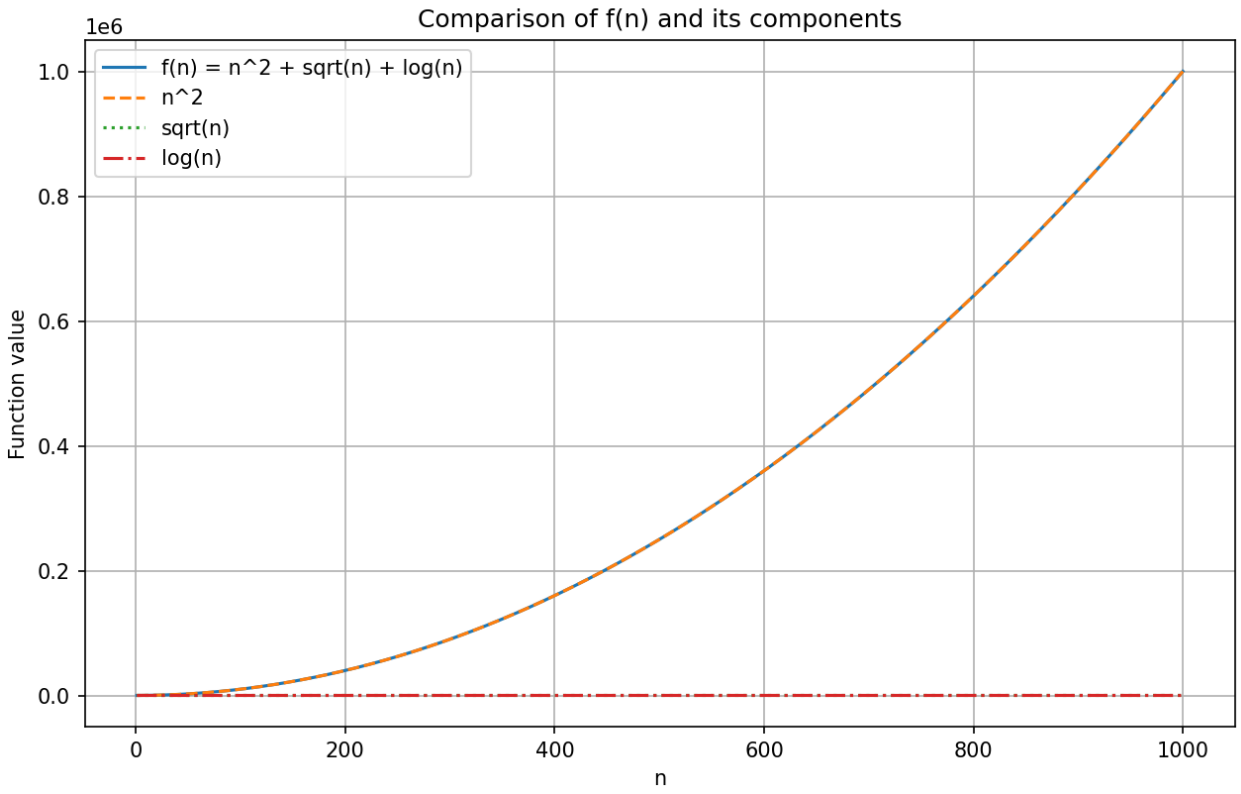
Graphical representation



4) f(n) = **$10n^2$ +sqrt(n) + log n**
Big O notation: O(**n^2**)

Analysis
1) n^2 is a quadratic term.
2) Sqrt(n) is a square root term, which grows slower than a linear term.

Log n is a logarithmic term, which grows slower than both linear and square root terms.
Among these, n^2 grows the fastest. So, the Big O complexity of this function is dominated by the n^2 term. Therefore, the Big O notation for f(n) is O(n^2)
Graphical representation

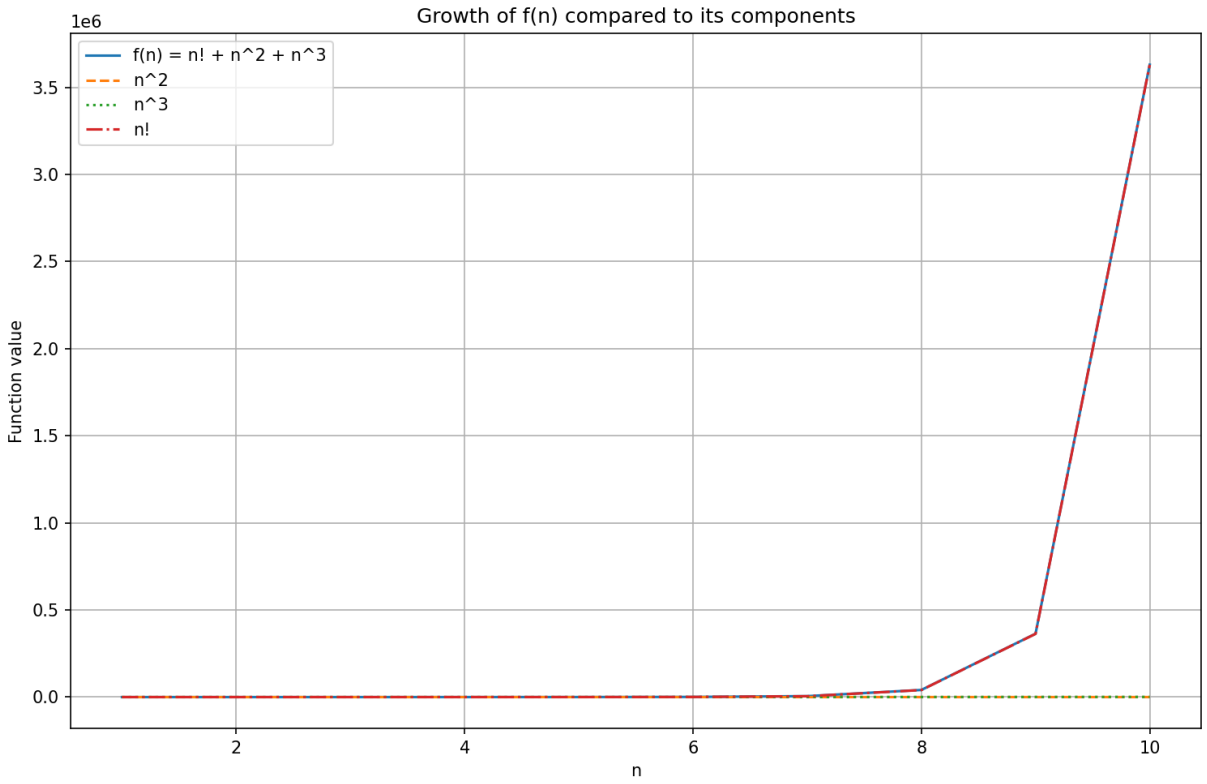


Example of big Omega Notation

1) f(n) = **$n! + n^2 + n^3$**
   Big Omega: Ω(**$n!$**)

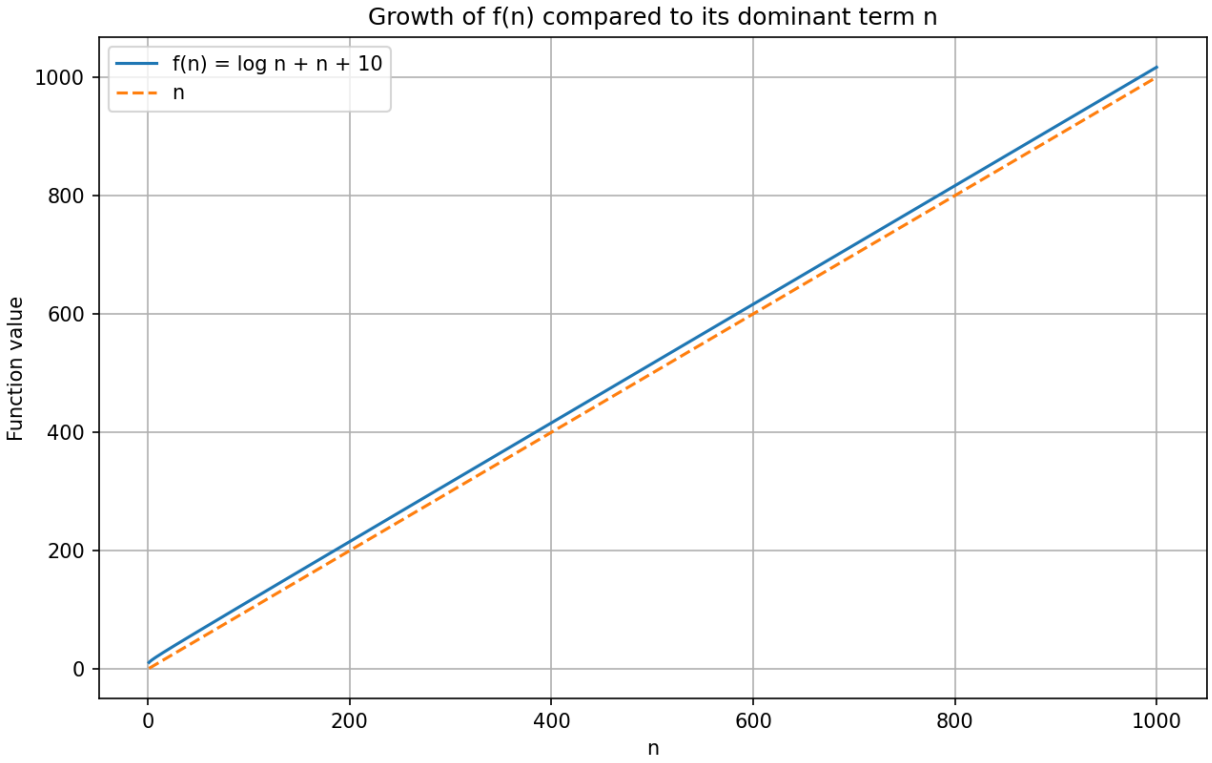
   Analysis
   1) The factorial term n! exhibits a super-exponential growth rate, surpassing polynomial, and linear growth significantly.
   2) There isn't a polynomial function that can serve as a lower bound for f(n) because the factorial function's growth rate exceeds any polynomial function.

   Graphical Representation



2) f(n) = **log n + n + 10**
Big Omega: Ω(**n**)

Analysis
The logarithmic term log(n) grows at a logarithmic rate, increasing slowly as n grows. It grows much slower than polynomial or exponential functions.
Graphical Representation



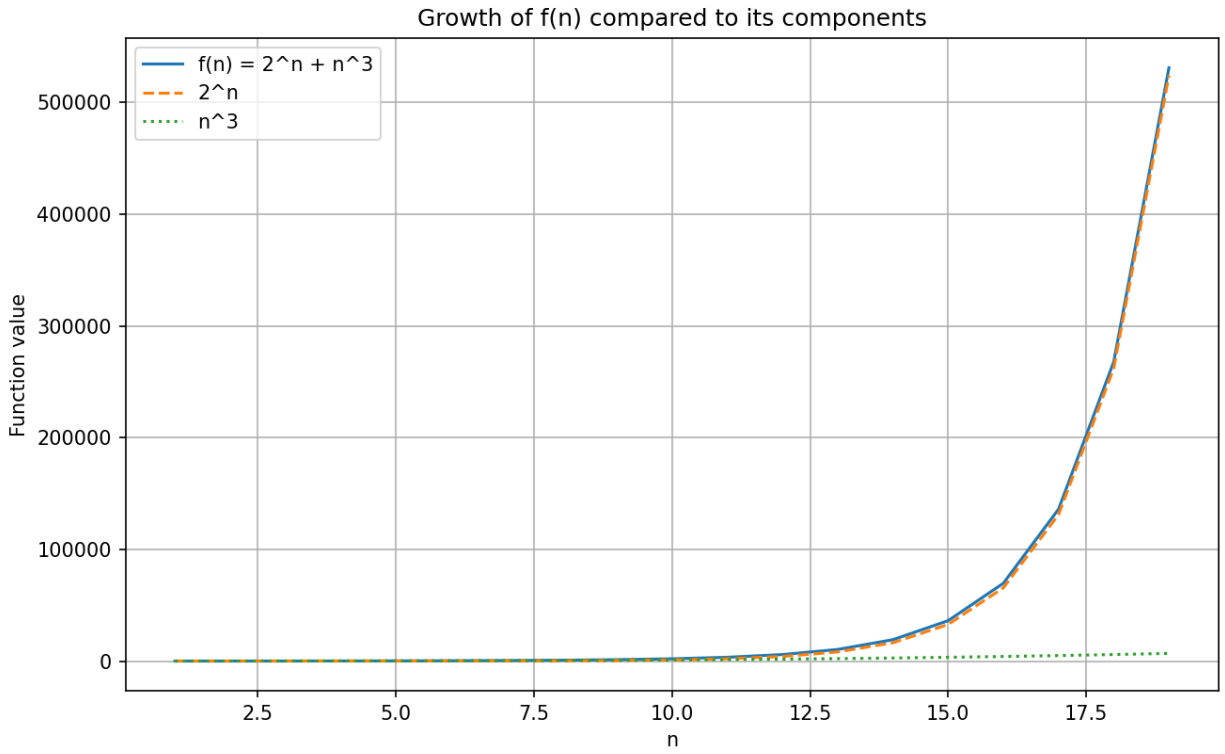
3) f(n) = $\mathbf{2^n + n^3}$

Big Omega: Ω($\mathbf{2^n}$)

Analysis

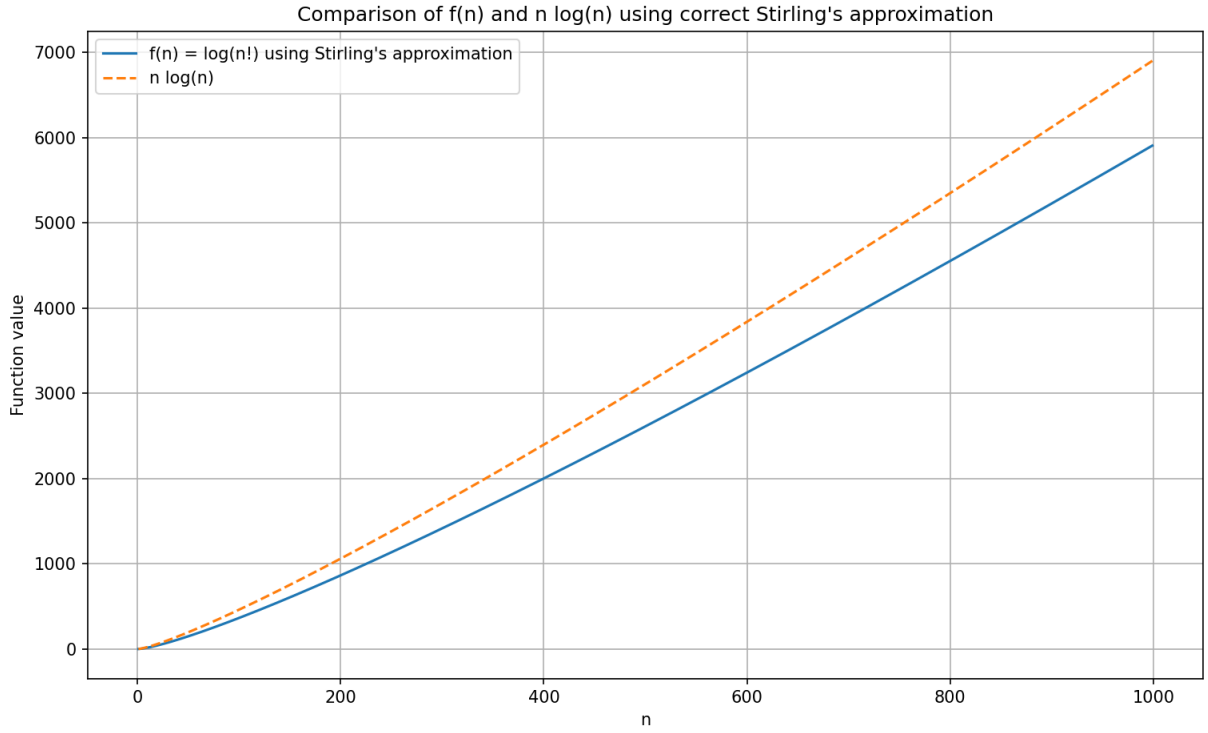
Even though the function has a cubic term, the exponential term 2^n grows faster. Thus, the function's growth is at least exponential, and it is Ω(2^n)

Graphical Representation



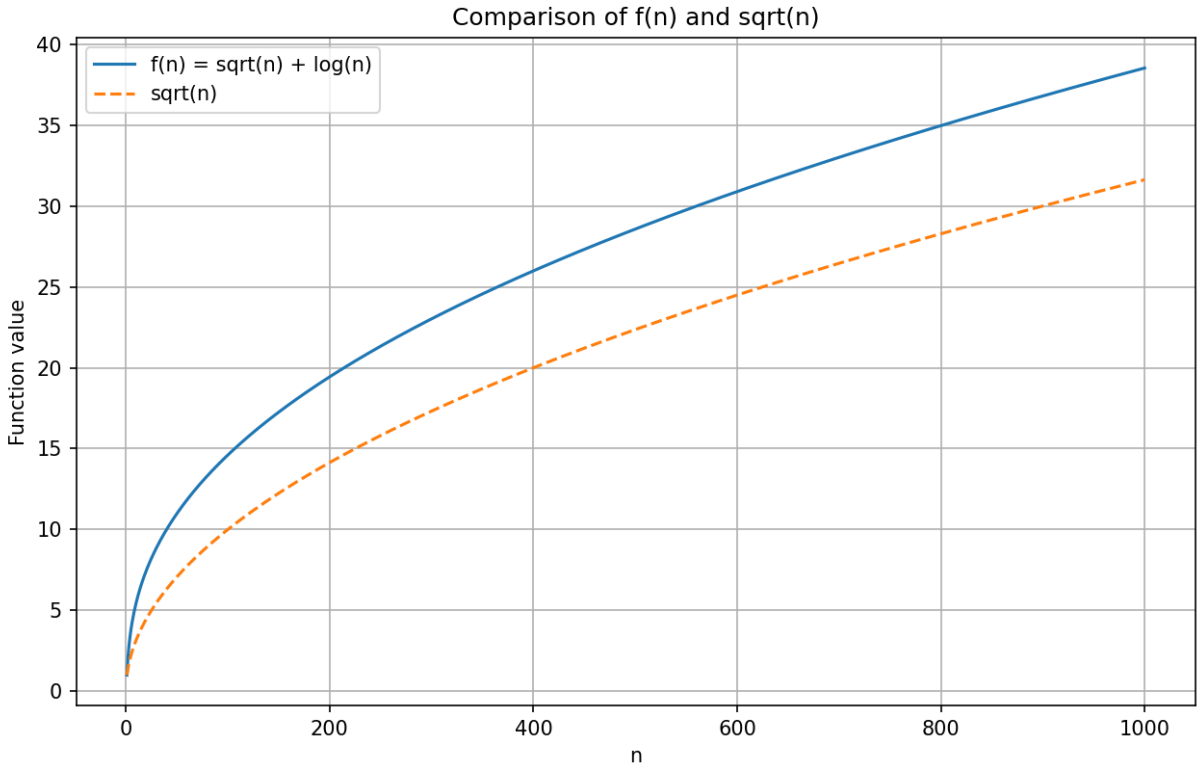
4) f(n) = **log(n!)**
Big Omega: Ω(**nlogn**)

Analysis
The logarithm function grows slower than the factorial, but log(n!) still represents a significant growth rate. It can be approximated using Stirling's approximation, which says that n! is approximately sqrt(2πn) * (n/e)^n. Taking the logarithm of this gives us something that grows on the order of n log(n).



5) f(n) = **sqrt(n) + logn**
Big Omega: Ω(**sqrt(n)**)

Analysis
As the n becomes larger, the dominant term in f(n) is sqrt(n) because its growth rate remains slower as compared to the logarithmic n for large n.
Graphical Representation:



Below are the common examples of Big O and Big Omega
Examples of Big O Notation (O):

1) O (1): It means that it takes a constant or same time to run an algorithm, regardless of the size of the input.
   Example: **Accessing the element in the array, push, and pop operation of stack.**
2) O(n): It means that the run time increases at the same pace as the input. Here n is the size of the input.
   Example: **Linear Search. So, which means by default it would have to loop the entire array to find the element.**
3) $O(n^2)$: It means that the calculation runs in quadratic time, which is the squared size of the input data.
   Example: **Bubble Sort**
4) $O(2^n)$: it means that the calculation grows with the exponential of the input.
   Example: **Fibonacci series without memorization.**
5) O (log n): it means that the running time grows in proportion to the logarithm of the input size, meaning that the run time barely increases as you exponentially increase the input. In such type of algorithms, typically, a portion of the input is discarded in each step, which is why the runtime increases very slowly compared to the input size.
   Example: **Binary Search**

Example of Big Omega Notation (Ω)

1) Ω (1): The algorithm has a lower bound that does not depend on the size of the input data. It takes a constant amount of time to complete, regardless of the input size.
   Example: **searching the element in the array, and if element found at position 1**
2) Ω (n): The best-case performance of the algorithm grows linearly with the input size. This means that in the best-case scenario, the number of operations increases directly proportionally to the size of the input.
   Example: **bubble sort**
3) Ω (n^2): The best-case performance grows at least as fast as the square of the input size. This indicates that even in the most favorable scenario, the algorithm performs a number of operations that are proportional to the square of the size of the input.
   Example: **Redundant sort**
4) Ω (n log n): The best-case performance of the algorithm is at least proportional to n log n. This complexity arises in algorithms that combine linear time operations with a logarithmic number of steps, often seen in efficient sorting algorithms.
   Example: **Quicksort**