**Algorithm:** a set of steps to solve a problem.

- Algorithms are essential for efficiently solving complex problems.

- They automate processes, making them more reliable and faster.

- Enable computers to perform tasks challenging for humans.

- Applied in mathematics, computer science, engineering, finance, and more.

- Optimize processes, analyze data, make predictions, and provide solutions.

**Way to write an Algo (examples):-**

Algorithm Sum(A, n)  
{  
 s=0;

for i=1 to n

s=s+A[i];

return s;

}

Algorithm Max(A,n)

{

Result= A[1]

for i=2 to n

if (A[i]>result) then result=A[i];

Return result;

}

**Algorithms as a Technology**

- Efficient algorithms optimize time and space, enhancing system performance.

- Algorithms are a critical technology alongside hardware advancements.

**Algorithm Analysis**

- Involves evaluating their efficiency and performance.

- Time complexity (how long an algorithm takes to run) and Space complexity (how much memory it uses).

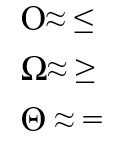
- Big O notation is commonly used to express algorithmic complexity.

- Efficiency is crucial for large datasets or resource-constrained environments.

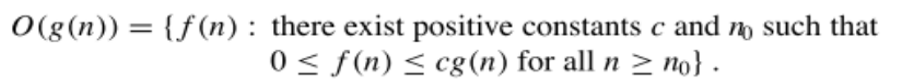
- Different algorithms may be better suited for specific types of problems or inputs.

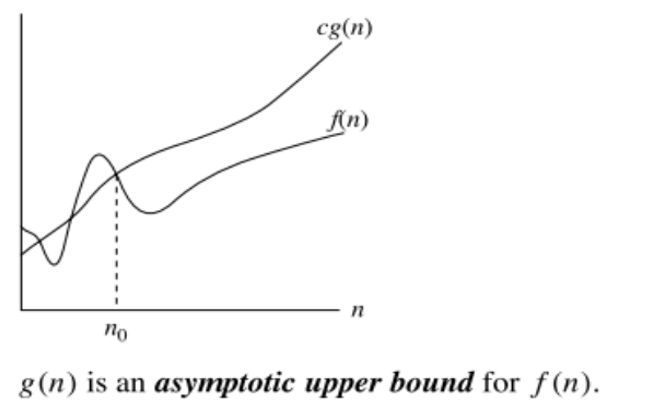
**Time Complexities:-**   
- ***O(1):*** Constant - Single or fixed number of statements, no loops involved. Example: Accessing an element in an array by index. Regardless of the array's size, accessing a specific element takes a constant amount of time.  
- ***O(log n):*** Logarithmic - Halves the problem size in each step (divide-and-conquer). Example: Binary search in a sorted array. With each comparison, the search space is halved, leading to a logarithmic time complexity.  
- ***O(n):*** Linear - Grows linearly with the input size; execution time scales linearly. Example: Iterating through an array to find a specific element. The time taken increases linearly with the size of the array since each element needs to be checked once.  
- ***O(n log n):*** Linearithmic - Slightly faster than quadratic time; common in efficient sorting. Example: Merge Sort or Heap Sort. These sorting algorithms have a time complexity that grows in proportion to the size of the input multiplied by the logarithm of the size of the input.  
- ***O(n^2):*** Quadratic - Execution time is a square of the input size; often seen in nested loop. Example: Selection Sort or Bubble Sort. These sorting algorithms involve nested iterations over the elements, resulting in a quadratic time complexity.  
- ***O(2^n):*** Exponential - Execution time grows exponentially with input size. Example: Recursive algorithm for generating all subsets of a set. The number of recursive calls doubles with each increase in the input size, resulting in exponential growth.

**Space Complexities:-**- ***O(1):*** Constant - Algorithms that use a fixed amount of memory regardless of input size. Example: Storing a fixed number of variables. The amount of memory used remains constant, regardless of the input size.  
- ***O(log n):*** Logarithmic - Require additional space that grows logarithmically with input size. Example: Recursive binary search. The space required increases logarithmically with the input size due to the recursive function calls.  
- ***O(n):*** Linear - Algorithms that consume space proportional to the input size. Example: Storing elements in an array. The space used is directly proportional to the input size, as each element needs to be stored.  
- ***O(n log n):*** Linearithmic - Algorithms that use space that grows in a linearithmic manner. Example: Merge Sort or Heap Sort (if performed in-place). These sorting algorithms require additional space that grows in a linearithmic manner.  
- ***O(n^2):*** Quadratic - Algorithms that use space proportional to the square of the input size. Example: Storing elements in a two-dimensional array. The space used increases quadratically with the input size for algorithms involving nested iterations.  
- ***O(2^n):*** Exponential - Use an amount of memory that \*2 with each addition to input size. Example: Recursive algorithm for generating all subsets of a set. The amount of memory used doubles with each additional element in the set, resulting in exponential growth.

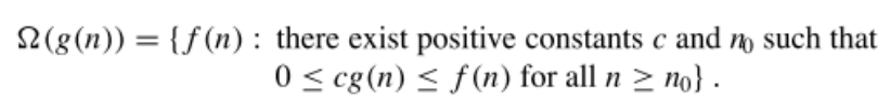
**Asymptotic notations** offer a way to analyze and compare algorithms by focusing on their efficiency as input sizes increase. They ignore machine-specific details, allowing a more universal understanding of algorithmic performance trends.

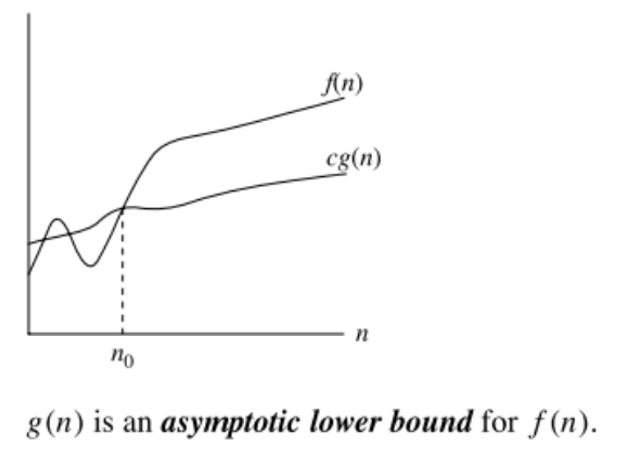
- ***Big-O (O-notation):*** Describes the worst-case time complexity of an algorithm. Simply a<=b

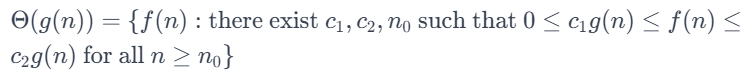


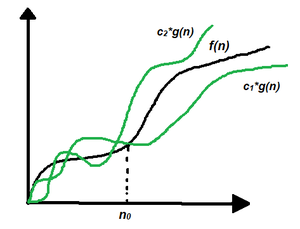


- ***Omega (Ω-notation):*** Best-case time complexity of an algorithm. Simply a>=b





- ***Theta (Θ-notation):*** Encloses the function from both above and below, used for average-case complexity. Simply a=b



**Analysis:-**

***Worst Case Analysis:***

- Determines the maximum time or space required by an algorithm for any input of size n.

- Provides an upper bound on the algorithm's performance, ensuring no input exceeds this bound.

- Example: In sorting algorithms, the worst-case scenario might involve sorting an array in descending order.

***Best Case Analysis:***

- Determines the minimum time or space required by an algorithm for any input of size n.

- Identifies the most favorable scenario for the algorithm's performance.

- Example: For sorting, the best case might involve sorting an already sorted array.

***Average Case Analysis:***

- Determines average time/space required by an algorithm over all possible inputs of size n.

- Offers a more realistic estimate of performance compared to worst or best cases alone.

- Example: In searching algorithms, consider a uniform distribution of target values across input data.