Overview:

Recurrence relations **are used to determine the running time of recursive programs** – recurrence relations themselves are recursive.

T(0) = Time to solve problem of size 0
T(n) = Time to solve problem of size n

There are many ways to solve a recurrence relation running time:
1) Back substitution
2) By Induction
3) Use Masters Theorem
4) Recursion tree

Examples of recurrence relations:

Time and Space complexity in Data Structure

What Is Time Complexity?

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input.

Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.

Time complexity is a type of computational complexity that describes the time required to execute an algorithm.

The time complexity of an algorithm is the amount of time it takes for each statement to complete.

As a result, it is highly dependent on the size of the processed data.

It also aids in defining an algorithm's effectiveness and evaluating its performance.

What Is Space Complexity?

When an algorithm is run on a computer, it necessitates a certain amount of memory space.

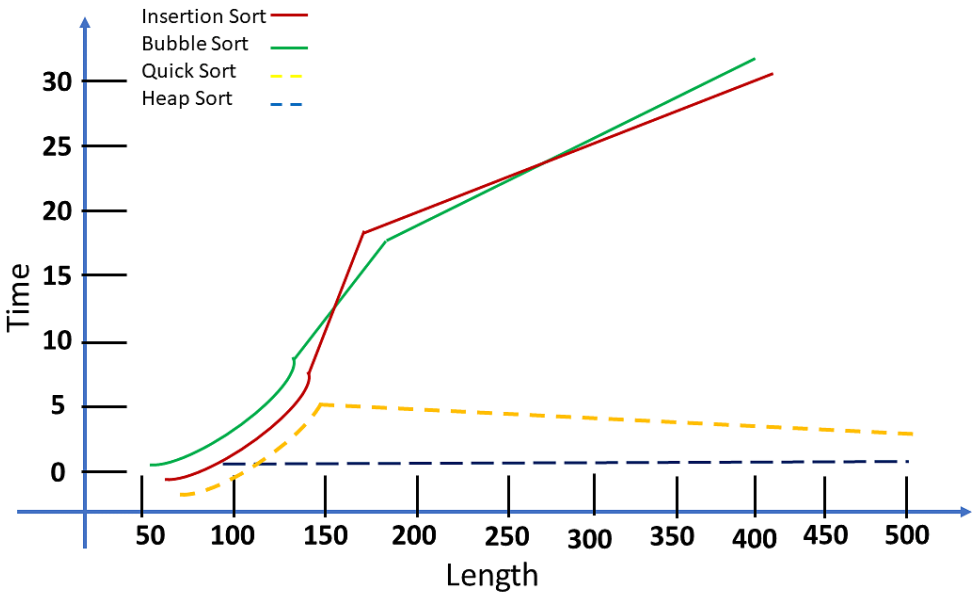
The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

The input size has a strong relationship with time complexity. As the size of the input increases, so does the runtime, or the amount of time it takes the algorithm to run.

Here is an example.

Assume you have a set of numbers S= (10, 50, 20, 15, 30)

There are numerous algorithms for sorting the given numbers. However, not all of them are effective. To determine which is the most effective, you must perform computational analysis on each algorithm.



Here are some of the most critical findings from the graph:

- This test revealed the following sorting algorithms: Quicksort, Insertion sort, Bubble sort, and Heapsort.
- Python is the programming language used to complete the task, and the input size ranges from 50 to 500 characters.
- The results were as follows: "Heap Sort algorithms performed well despite the length of the lists; on the other hand, you discovered that Insertion sort and Bubble sort algorithms performed far worse, significantly increasing computing time." See the graph above for the results.
- Before you can run an analysis on any algorithm, you must first determine its stability. Understanding your data is the most important aspect of conducting a successful analysis.

What Are Asymptotic Notations?

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows.

This is also referred to as an algorithm's growth rate.

You can't compare two algorithms head to head. It is heavily influenced by the tools and hardware you use for comparisons, such as the operating system, CPU model, processor generation, and so on. Even if you calculate time and space complexity for two algorithms running on the same system, the subtle changes in the system environment may affect their time and space complexity.

As a result, you compare space and time complexity using asymptotic analysis. It compares two algorithms based on changes in their performance as the input size is increased or decreased.

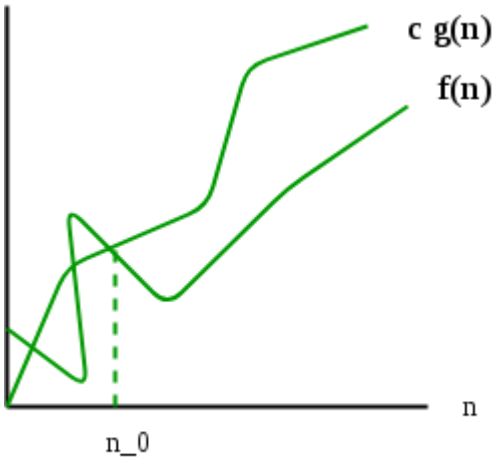
Asymptotic notations are classified into three types:

1. Big-Oh (O) notation
2. Big Omega (Ω) notation
3. Big Theta (Θ) notation

2. Big-O Notation (O-notation):

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

The maximum time required by an algorithm or the worst-case time complexity.



f(n) = O(g(n))

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term.

For example, if an algorithm is O(n), it means that its runtime grows linearly with the input size, which means if the input size doubles, the runtime will roughly double as well.

O(1): Constant time complexity, meaning the algorithm's runtime remains the same regardless of the input size.

O(log n) - Logarithmic Time: Algorithms with O(log n) time complexity have their execution time increase logarithmically with the input size. They are often associated with efficient binary search algorithms.

O(n): Linear time complexity, where the runtime grows linearly with the input size.

O(n log n) - Linearithmic Time: This complexity arises in algorithms like merge sort and quicksort. It's more efficient than O(n^2) for large inputs.

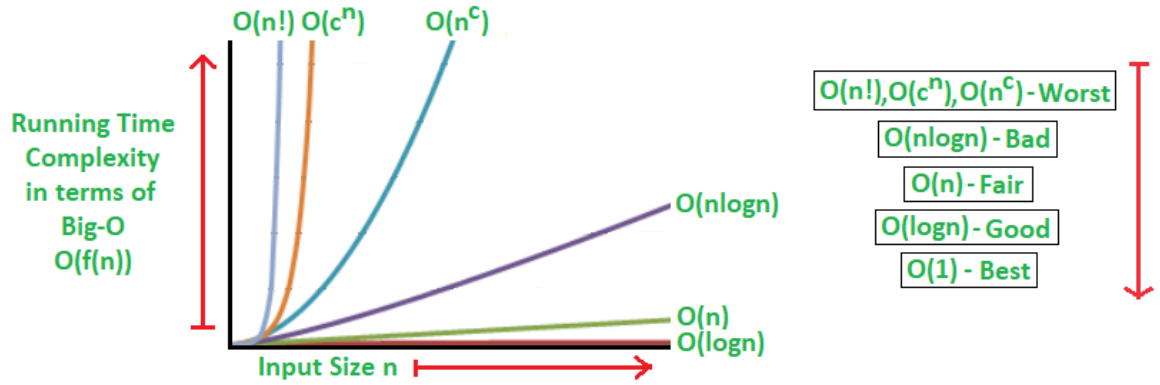
O(n^2) - Quadratic Time: Algorithms with O(n^2) time complexity have their execution time increase quadratically with the input size.

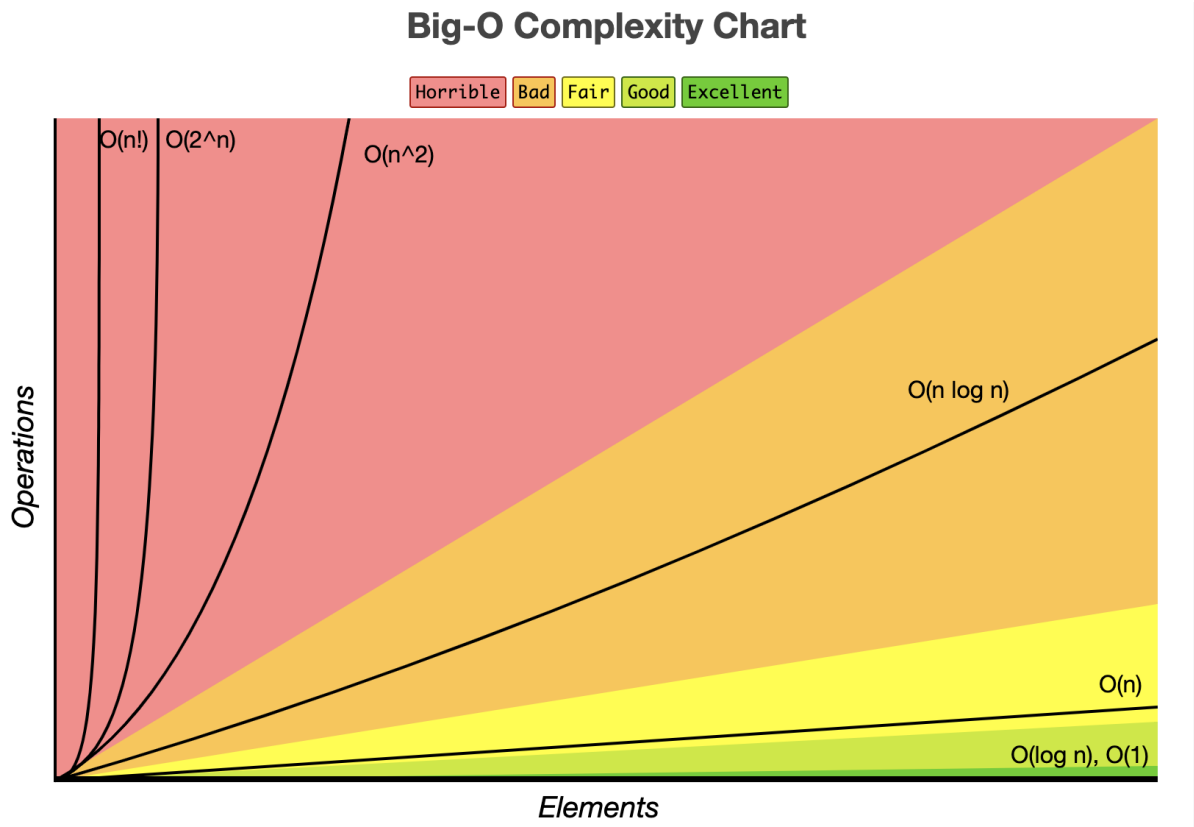
```
for i from 1 to n:
    for j from 1 to n:
```

O(n^3), pronounced as "big O of n cubed" or "order n cubed," represents an algorithmic time complexity where the execution time of an algorithm grows at a rate proportional to the cube of the input size (n).

```
for i from 1 to n:
    for j from 1 to n:
        for k from 1 to n:
```

O(n!) - Factorial Time: Algorithms with O(n!) time complexity have the slowest possible growth, with execution time increasing by a factor of "n!" (n factorial) as the input size increases.





When analyzing the time complexity of algorithms using Big O notation, it's common to ignore lower-order terms and constants. This simplification helps focus on the most significant factors affecting the algorithm's growth rate as the input size increases.

Suppose you have an algorithm with a time complexity expressed as:

O(3n^2 + 5n + 7)

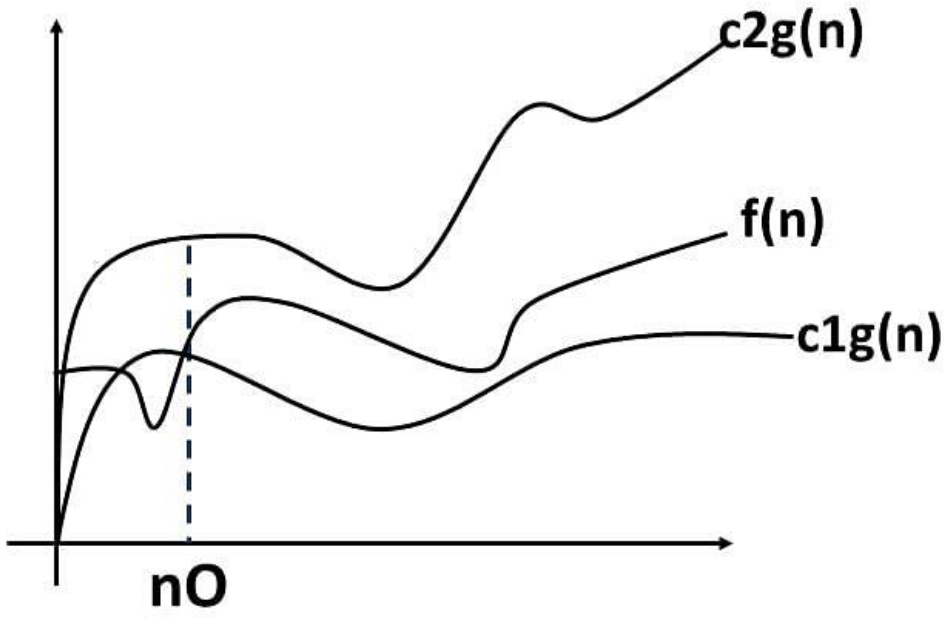
To ignore lower-order terms and constants, we focus on the highest-order term, which is n^2, and ignore the coefficients and lower-order terms. So, after simplification, the time complexity becomes:

O(n^2)

Definition: A function f(n) is said to belong to the set O(g(n)) if there exist positive constants c and n0 such that for all n greater than or equal to n0, 0 ≤ f(n) ≤ cg(n). In other words, f(n) is upper-bounded by cg(n) for large enough n.

1. Theta Notation (Θ-Notation):

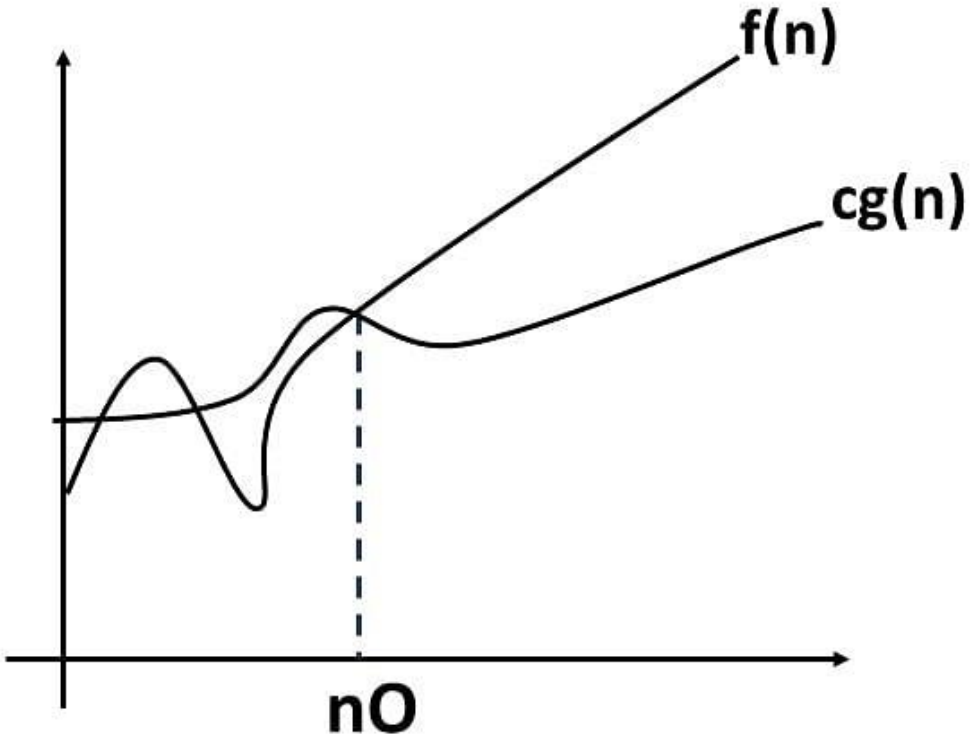
Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



$f(n) = \theta(g(n))$

For example, if an algorithm is Θ(n), it means that its runtime grows exactly linearly with the input size, and the runtime can't grow faster or slower than that.

3. 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.



$$f(n) = \Omega(g(n))$$

For example, if an algorithm is Ω(n^2), it means that its runtime grows at least as fast as the square of the input size.

Substitution Method in Data Structure

The substitution method is a mathematical technique used in the analysis of algorithms and data structures to determine their time complexity.

Master Theorem:

Applicability: The Master Theorem is applicable to a specific class of recurrence relations that follow a particular form. **It is most commonly used for divide-and-conquer algorithms.**

Substitution Method:

Applicability: The substitution method is a more general approach and can be used to analyze a wider range of recurrence relations, **including those that do not fit the Master Theorem's specific cases.**

Recurrence Tree Method:

Applicability: The recurrence tree method is particularly **useful for visualizing the recursive structure of an algorithm and understanding how it breaks down into subproblems**.