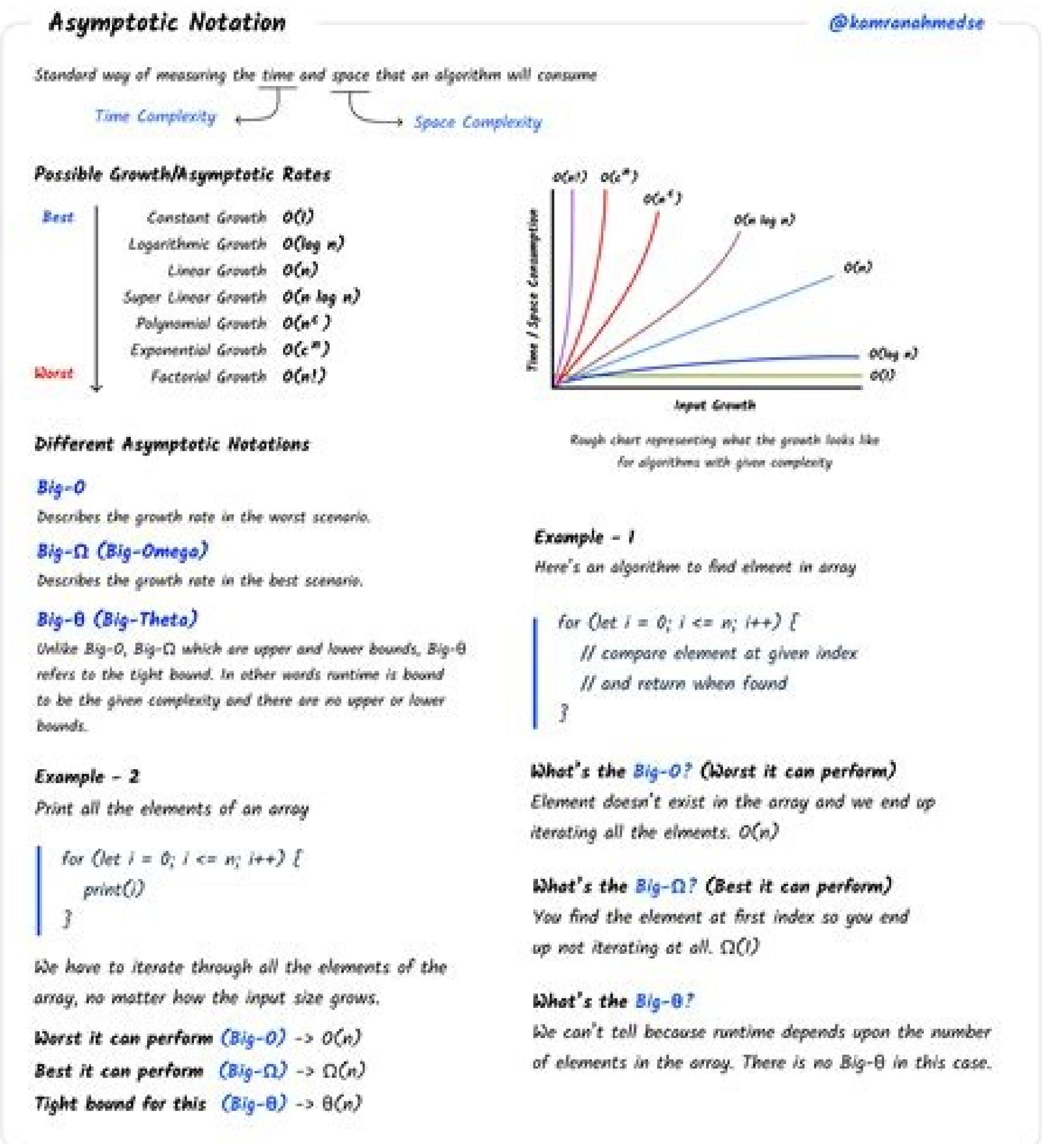
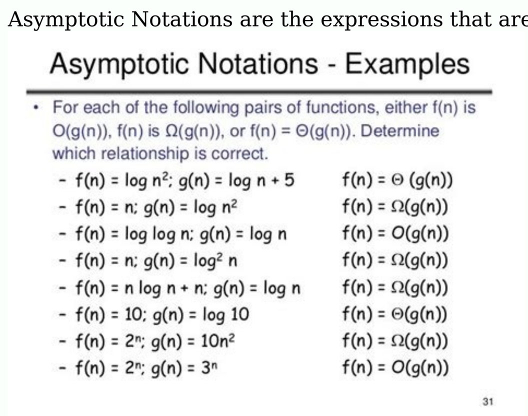
Asymptotic notation in data structure pdf

Asymptotic notation in data structure with example. Why we need asymptotic notation. What is asymptotic notation. What is asymptotic notation in data structure.

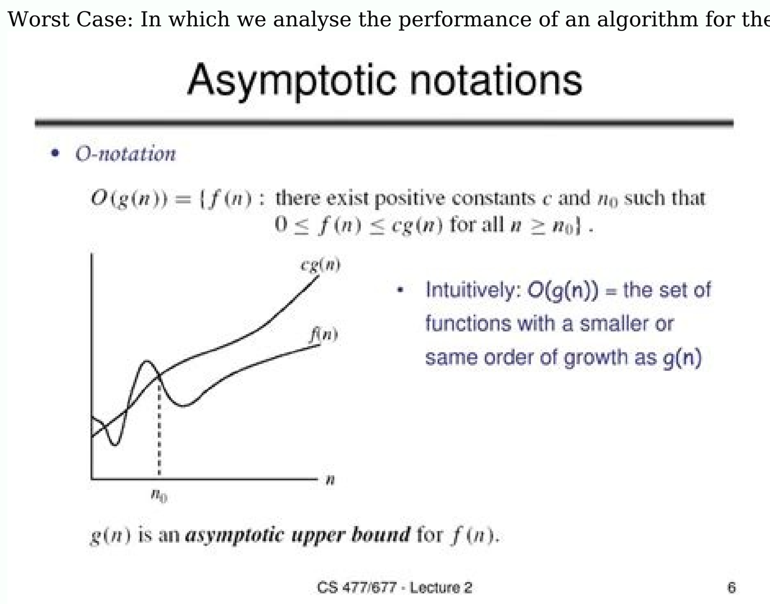
The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm.
The efficiency is measured with the help of asymptotic notations. An algorithm may not have the same performance for different types of inputs. With the increase in the input size, 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 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. Big-O gives the upper bound of a function 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 analyze 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.



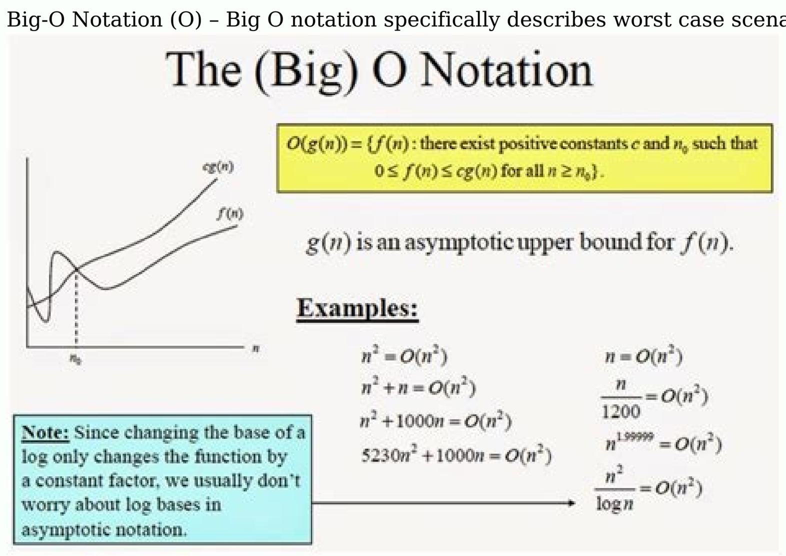
Asymptotic Notations are the expressions that are used to represent the complexity of an algorithm. As we discussed in the last tutorial, there are three types of analysis that we perform on a particular algorithm. Best Case: In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.



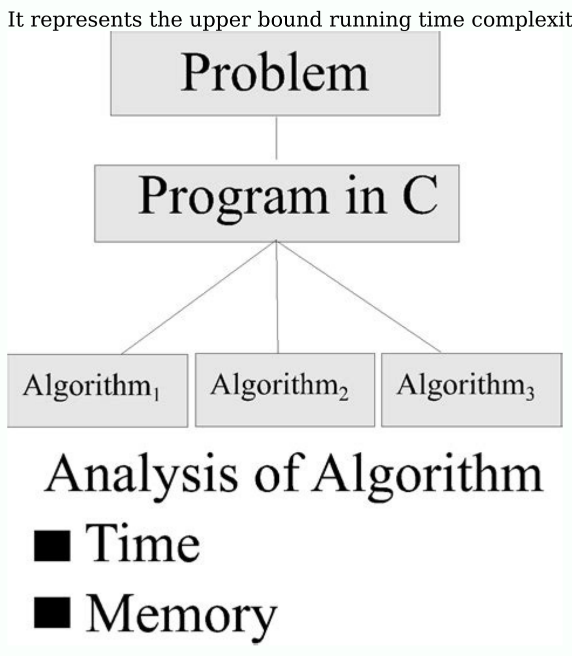
Worst Case: In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space. Average Case: In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case. Types of Data Structure Asymptotic Notation 1.



Big-O Notation (Ο) – Big O notation specifically describes worst case scenario. 2. Omega Notation (Ω) – Omega(Ω) notation specifically describes best case scenario. 3. Theta Notation (θ) – This notation represents the average complexity of an algorithm. Big-O Notation (Ο) Big O notation specifically describes worst case scenario.



It represents the upper bound running time complexity of an algorithm. Lets take few examples to understand how we represent the time and space complexity using Big O notation.



O(1) Big O notation O(1) represents the complexity of an algorithm that always execute in same time or space regardless of the input data. O(1) example The following step will always execute in same time(or space) regardless of the size of input data. Accessing array index(int num = arr[5]) O(n) Big O notation O(N) represents the complexity of an algorithm, whose performance will grow linearly (in direct proportion) to the size of the input data. O(n) example The execution time will depend on the size of array. When the size of the array increases, the execution time will also increase in the same proportion (linearly) Traversing an array O(n^2) Big O notation O(n^2) represents the complexity of an algorithm, whose performance is directly proportional to the square of the size of the input data. O(n^2) example Traversing a 2D array Other examples: Bubble sort, insertion sort and selection sort algorithms (we will discuss these algorithms later in separate tutorials) Similarly there are other Big O notations such as: logarithmic growth O(log n), log-linear growth O(n log n), exponential growth O(2^n) and factorial growth O(n!). If I have to draw a diagram to compare the performance of algorithms denoted by these notations, then I would draw it like this: O(1) < O(log n) < O (n) < O(n log n) < O(n^2) < O (n^3)< O(2^n) < O(n!) Omega Notation (Ω) Omega notation specifically describes best case scenario. It represents the lower bound running time complexity of an algorithm. So if we represent a complexity of an algorithm in Omega notation, it means that the algorithm cannot be completed in less time than this, it would at-least take the time represented by Omega notation or it can take more (when not in best case scenario). Theta Notation (θ) This notation describes both upper bound and lower bound of an algorithm so we can say that it defines exact asymptotic behaviour. In the real case scenario the algorithm not always run on best and worst cases, the average running time lies between best and worst and can be represented by the theta notation.
ReadDiscussCoursesPracticeImprove Article Save Article Like Article In mathematics, asymptotic analysis, also known as asymptotics, is a method of describing the limiting behavior of a function. In computing, asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance based on the input size. For example, the running time of one operation is computed as f(n), and maybe for another operation, it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is small in value.Usually, the analysis of an algorithm is done based on three cases:Best Case (Omega Notation (Ω))Average Case (Theta Notation (Θ))Worst Case (O Notation(O))All of these notations are discussed below in detail:Omega (Ω) Notation:Omega (Ω) notation specifies the asymptotic lower bound for a function f(n). For a given function g(n), Ω(g(n)) is denoted by:Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 ≤ c*g(n) ≤ f(n) for all n ≥ n0}. This means that, f(n) = Ω(g(n)), If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).Graphical representationFollow the steps below to calculate Ω for a program:Break the program into smaller segments.Find the number of operations performed for each segment(in terms of the input size) assuming the given input is such that the program takes the least amount of time.Add up all the operations and simplify it, let's say it is f(n).Remove all the constants and choose the term having the least order or any other function which is always less than f(n) when n tends to infinity, let say it is g(n) then, Omega (Ω) of f(n) is Ω(g(n)).Omega notation doesn't really help to analyze an algorithm because it is bogus to evaluate an algorithm for the best cases of inputs.Theta (Θ) Notation:Big-Theta(Θ) notation specifies a bound for a function f(n). For a given function g(n), Θ(g(n)) is denoted by:Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1*g(n) ≤ f(n) ≤ c2*g(n) for all n ≥ n0}. This means that, f(n) = Θ(g(n)), If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or above c1*g(n) and below c2*g(n).Graphical representationFollow the steps below to calculate Θ for a program:Break the program into smaller segments.Find all types of inputs and calculate the number of operations they take to be executed. Make sure that the input cases are equally distributed.Find the sum of all the calculated values and divide the sum by the total number of inputs let say the function of n obtained is g(n) after removing all the constants, then in Θ notation, it's represented as Θ(g(n)). Example: In a linear search problem, let's assume that all the cases are uniformly distributed (including the case when the key is absent in the array). So, sum all the cases when the key is present at positions 1, 2, 3, ……, n and not present, and divide the sum by n + 1.Average case time complexity = ⇒ ⇒ ⇒ Since all the types of inputs are considered while calculating the average time complexity, it is one of the best analysis methods for an algorithm.Big – O Notation:Big – O (O) notation specifies the asymptotic upper bound for a function f(n). For a given function g(n), O(g(n)) is denoted by:O (g(n)) = {f(n): there exist positive constants c and n0 such that f(n) ≤ c*g(n) for all n ≥ n0}. This means that, f(n) = O(g(n)), If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).Graphical representationFollow the steps below to calculate O for a program:Break the program into smaller segments.Find the number of

operations performed for each segment (in terms of the input size) assuming the given input is such that the program takes the maximum time i.e the worst-case scenario.Add up all the operations and simplify it, let's say it is f(n).Remove all the constants and choose the term having the highest order because for n tends to infinity the constants and the lower order terms in f(n) will be insignificant, let say the function is g(n) then, big-O notation is O(g(n)).It is the most widely used notation as it is easier to calculate since there is no need to check for every type of input as it was in the case of theta notation, also since the worst case of input is taken into account it pretty much gives the upper bound of the time the program will take to execute.Last Updated : 28 Oct, 2022Like Article Save Article