**FORMAL ASYMPTOTIC NOTATIONS OF COMPLEXITY BOUNDS WITH EXPLANATIONS**

*Submitted in Partial Fulfillment of the Second Continuous Assessment*

**Bachelor of Technology**

In

Computer Science and Engineering (AI&ML)

*4th semester*

**NAME: ADITIYA KUMAR**

**UNIVERSITY ROLL NO:**

**UNIVERSITY REGISTRATION NO:**

**PAPER NAME: DESIGN AND ANALYSIS OF ALGORITHMS**

**PAPER CODE: PCC-CS-404**



*Under the guidance of*

**Paper Faculty**

**Name: Sourav Das**

**Designation: Assistant Professor**

**Department of CSE (AI&ML)**

**Future Institute of Technology**

**March 2023.**

* ABSTRACT:

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.

* INTRODUCTION:

We try to determine the estimate of runtime and space time complexities for each algorithm that we encounter. However, the runtime complexity of the same algorithm may differ depending upon the different environment.

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input.

* PROBLEM DEFINITION:
* Asymptotic Notation means limiting the behavior in terms of computational complexity for any growing algorithm. The computation complexity can further be explained in two ways (a) time complexity (b) space complexity.
* Asymptotic Notation give us an idea about how good a given algorithm is compared to some other algorithm.
* Asymptotic Notation of an algorithm is a mathematical representation of its complexity.
* 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.
* Primarily there are three types of widely used asymptotic notations.

1. Big oh notation (0).
2. Big omega notation (Ω).
3. Big theta notation (Ṍ).

* DISCUSSION:
* RULE OF THUMB: (anything which is obvious and can be determined).

1. Simple programs can be studied for complexity by observing the number of loops.
2. Given a program with or without a series of loops the behavior for processing each loop is important because the slowest among all the loops essentially determines the computational complexity growth for that program.
3. In a program tow nested loops followed by a single loop separately will yield the same asymptotic behavior when compared to another program having three nested loops.
4. BIG OH COMPLEXITY CHART:



**1.BIG OH NOTATION (O):**

Big oh notation is used to describe asymptotic upper bound.

Big-oh notation always indicates the maximum time required by an algorithm for all input values.

Mathematically, if f(n) describes running time of an algorithm, f(n) is O(f(n)) iff there exist positive constants C and no such that

O<=f(n)<=Cg(n).

For all n>=no used to give upper bound on a function.

* DEFINITION:

 Let t and g be two functions that map a set of natural numbers to a set of positive real numbers, that is, t: N→ R. Let O (g) is the set of all functions with a similar rate of growth. The relation t (n) = O (g (n)) holds true, if there exist two positive constants c and no such that t(n) ≤ c\*g(n). The function (n) is said to be in O (g (n). This is denoted as t (n) = O (g (n)) or simply as t (n) =

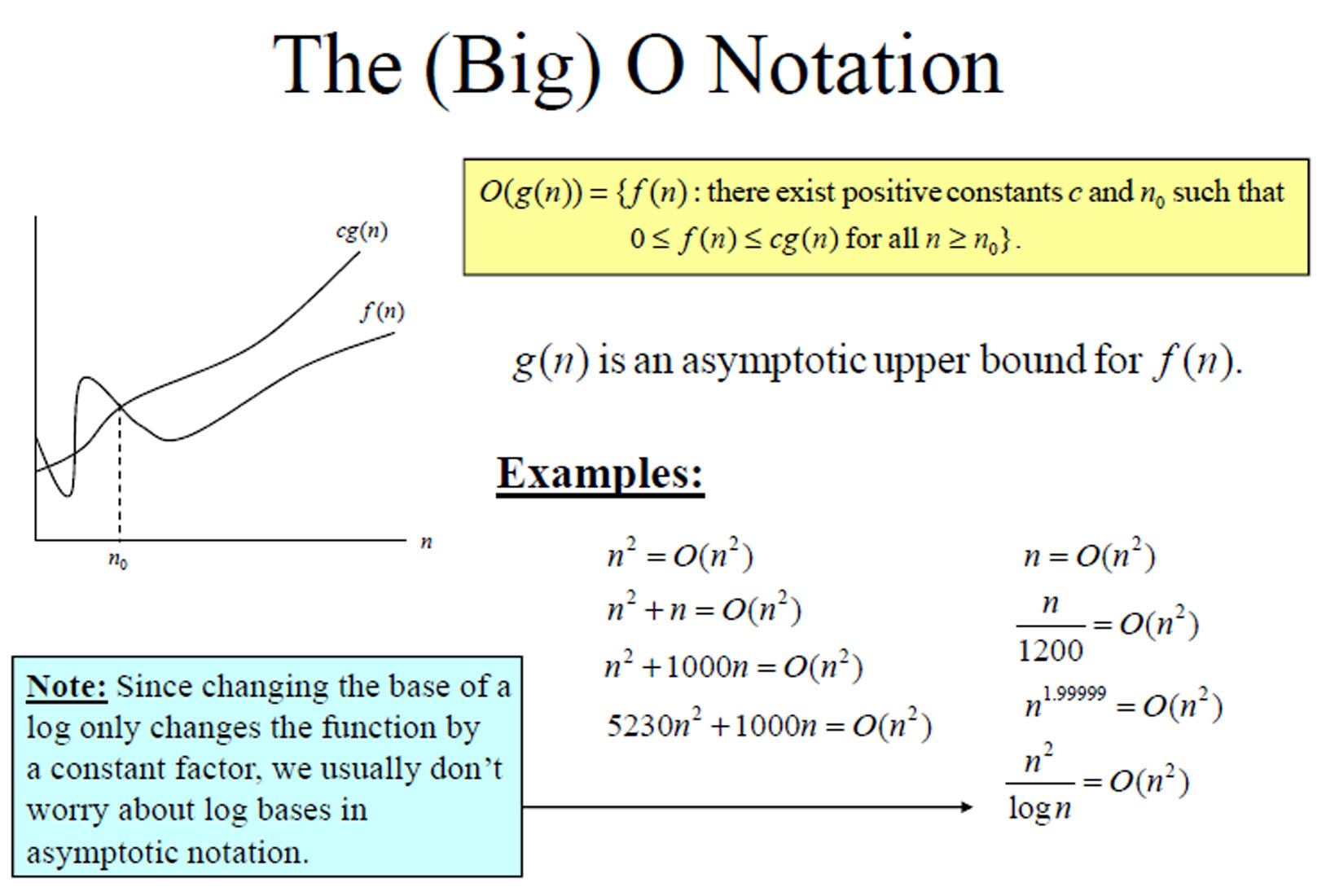
O (g(n)). This implies that t(n) never takes more than approximately g(n) operations; that is, t(n) is in the order of g(n) or simply a function with a growth rate that is less than or equal to that of g(n). This implies that grows  
at a slower rate than a constant time g(n) for all the values of a larger input of size 'n'. This is shown in fig 1.

FIG-1

* PROPERTIES OF BIG –OH NOTATION:

The following are some of the properties of big oh:  
1. For big-Oh, only the dominating summand matters. For example, O (n++ n2+64) = O (n1). It can be observed that all terms other than the highest degree are ignored.  
2. In addition, in the big-Oh notation, constant factors are not significant. For example, O (3n3) = O (n3). In general, O (k g(n)) = O(g(n)), where k + 0.  
.  
3. Big-Oh can be used to express tight bounds. A bound is called a tight bound or least upper bound if the difference between the actual and bound functions is a constant. For example, n2 cannot be expressed as O (n3). It should only be expressed as O (n2), as it is the best fit. In this case, it is called a tight fit or the least upper bound.

* EXAMPLES:

Example - Let t(n) = 3n^3 for an algorithm. Prove that t (n) of the algorithm is in O (n^3).

SOLUTION- The definition of the big-Oh notation is that t (n) ≤ cx g(n). In order to prove that t(n) is in O(g(n)), where g(n) is n3, one has to show that 3n^3 ≤ cn^3 holds good for a positive number c and for sufficiently large values of n. It can easily be seen that this condition holds when c≥ 4. Therefore, t is O(g) or in other words, the algorithm is O(n^3).

**2.BIG-OMEGA NOTATION:**

Big-omega notation is used to describe asymptotic lower bound.

Big-omega notation always indicates the minimum time required by an algorithm for all input values.

Mathematically, if f (n) define running time of an algorithm, f(n) is said to be Ω(g(n)) if there exists positive constants c and no such that

O<=Cg (n) <=f (n)

For all n>=no, used to give lower bound.

* DEFINITION:

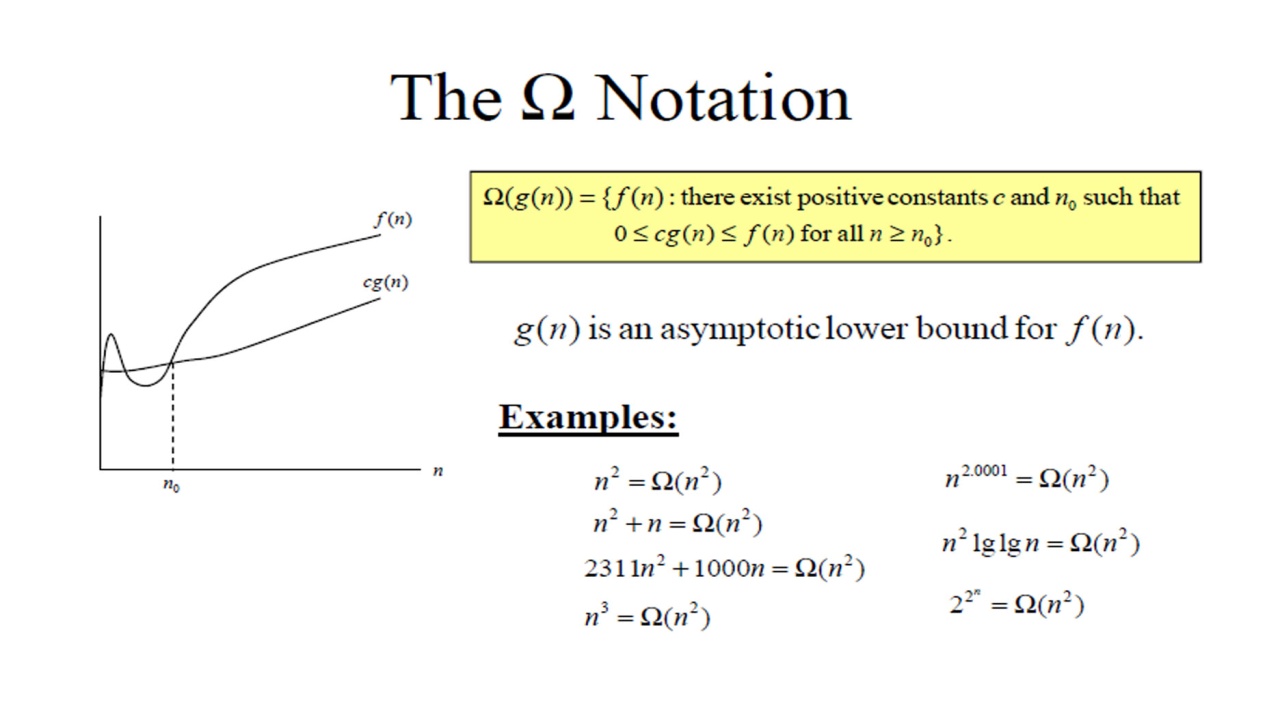
Let t and g be two functions that map a set of natural numbers to a set of positive real numbers, that is, t: N→ Ro. Let Ω(g) be the set of all those functions that have a similar rate of growth. The relation t(n) =Ω(g(n)) holds good if there exist two positive constants c and no such that t(n) > =c\*g(n). Thus, the function t (n) is said to be in Ω(g(n), which can be represented as t(n) = Ω(g(n)) or simply as t(n) = Ω(g(n)).  
This notation implies that t (n) grows at a faster rate than a constant time g (n) for a suf- ficiently large n. The big-omega notation, which is the lower bound of an algorithm, is illustrated in Fig.2.

FIG 2.

* PROPERTIES OF BIG-OMEGA NOTATION:

1. If f(n) is Big-Oh(g(n)), then a\*f(n) is Big-Oh(g(n))
2. If f(n) is Omega(g(n)), then a\*f(n) is Omega(g(n)).

* EXAMPLES:

If the relation t(n) = 6n2 + 7n + 8 holds, prove that t(n) is not in 2(n^3).

SOLUTION: To prove t(n) is in 2(n^3), one has to prove the following: 6n2+7n+8>cxn3 However, this cannot be proved as there is no positive number 'c' for which this condition holds good. Therefore, t (n)  
(n^3).

**3.BIG- THETA NOTATION:**

Big- theta notation is used to describe asymptotic average bound.

Big- theta notation always indicates the average time required by an algorithm for all input value.

Mathematically, let f (n) define running time of an algorithm, f (n) is said t be Ṍ(g(n)) iff f (n) is O(g(n) and f(n) is Ω(g(n)

O<=f (n)<=C1g(n) for all n>=no…….1

O<=C2g (n)<=f(n) for all n>=no…….2

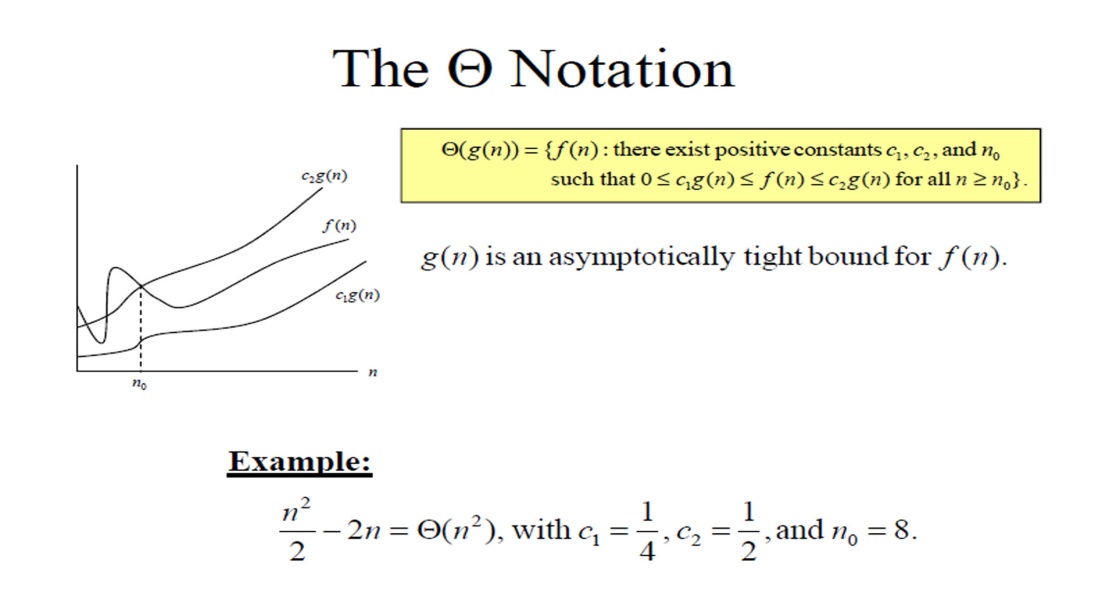
Merging both the equations, we get:-

O<=C2g (n)<=f (n)<=C1g (n) for all n>=n0

This equation simply means there exist positive constants C1, C2 such that f (n) is sandwiched between C2g (n) and C1g (n).

* DEFINITION:

Let t and g be two functions that map a set of natural numbers to a set of positive real numbers. If the relationship c1 g(n) ≤f(n) ≤ c2 g(n) holds good for all n ≥ no and for constants c, and c2, then t(n) can be denoted as t(n) = (g(n)).  
This is equivalent to saying that t(n) grows at the same rate as a constant time g(n) for all sufficiently large values of n. The notation is shown in Fig.3.



* PROPERTIES:

 The following are the properties of the big-theta notation:

1. If t (n) = O (g (n)) and g(n) = O(t(n)), then t(n) = O(n).  
2. For any polynomial of the order of k, one can show that t(n) is in O(n).  
Thus, asymptotic notations are helpful in representing the order of growth of an algorithm. The following examples illustrate the use of asymptotic notations to understand the behavior of an algorithm.

* EXAMPLES:

 Suppose that an algorithm takes eight seconds to run on an input size n = 12. Estimate the instances that can be processed in 56 seconds. Assume that the algorithm complexity is O (n).

SOLUTION: Assume that the time complexity is O(n), then cn≈ 8 seconds. Here, the instance n is given as 12. Therefore, 12c = 8; hence, c = 8/12 = 2/3.  
The problem is to determine the value of n that can be processed in 56 seconds. This implies that cx n = 56.  
The value of c has already been determined as 2/3. Therefore, 2/3 x n = 56. This implies = 56 x 3/2 = 84.  
that n =  
Hence, the maximum input that is possible is 84.

* CONCLUSION:

In conclusion, we can say that asymptotic notations can be used to simply perform an analysis of the run-time behavior of an algorithm. Using these, we represent the upper bound or lower bound of run-time in mathematical equations and thus help us perform our task with the best efficiency and less effort.

