UNIT- 2 ASYMPTOTIC BOUNDS

Structure

2. 0 INTRODUCTION

In the last unit, we introduced a survey of common running time complexities and highlighted several problems with different asymptotic bounds. Algorithmic complexity is an important area in computer science. If we know complexities of different algorithms then we can easily answer the following questions-

- How long will an algorithm/ program run on an input?
- How much memory will it require?
- Is the problem solvable?

The above mentioned criterions are used as thebasis of the comparison among different algorithms. With the help of algorithmic complexity, programmers improve the quality of their code using relevant data structures. To measure the efficiency of a code/ algorithm, asymptotic notations are normally used.

Asymptotic notations are the mathematical notations that estimate the time or space complexity of an algorithm or program as function of the input size. For example, the best case running time of a function that sorts a list of numbers using bubble sort will be linear i.e., O(n). On the contrary, the worst case running time will be $O(n^2)$. So, we can say that the bubble sort takes T(n) time, where, $T(n)=O(n^2)$. The asymptotic behavior of a function f(n) indicate the the growth of f(n) as n gets very large. The small values of n are generally ignored as we are interested to know how slow the program or algorithm will be on large input. It is stated in literature- the slower asymptotic growth rate, the better the algorithm performance. As per this measurement, a linear algorithm (i.e., f(n)=d*n+k) is always asymptotically better than a quadratic one (e.g., f(n)=c*n^2+q) for any positive value of c, k, d, and q. To understand concepts of asymptotic notations, you will be given a idea of lower bound, upper bound, and an average bound. Mathematical induction plays an important role in computing the algorithms' complexity. Using the mathematical induction, problem is converted in the form mathematical expression which is solved to find the time complexity of algorithm. Further to rank algorithms in increasing or decreasing order asymptotic notations such as big oh, big omega, and big theta are used.

The focus of the unit is to define these bounds mathematically and discuss some useful theorems related to these bounds.

2. 1 Objectives

After studying this unit, you will be able to learn-

- Some well mathematical functions & notations
- Principle of mathematical induction
- Mathematical Definition of Asymptotic bounds
- Worst case, best case, and average case complexities
- Comparative analysis of different types of algorithms

2. 2 Some useful Mathematical Fnctions& Notations

To have completeness of the unit, in this section some useful mathematical functions and notations are provided.

2.2.1 Summation & Product

Summation:

Suppose we are having a sequence of numbers1 ,2,....n where n is a integer number, the finite sum of these sequences, i.e. 1+ 2+......n can be denoted as:
$\sum_{i=1}^{n} i$, where Σ is called sigma symbol
$\sum_{i=1}^{n} i$ is in arithmetic series and has the values

Sum of sequences

(i) $\sum_{i=1}^{n} i = 1 + 2 + \cdots n = \frac{n(n+1)}{2}$

(ii) $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots n^2 = \frac{n(n+1)(2n+1)}{6}$

(iii) $\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + \cdots n^3 = \frac{n^2(n+1)^2}{4}$

Product
The expression
$1 \times 2 \times \ldots \times n$
can be denoted in shorthand as

$$\prod_{i=1}^{n} i$$

2.2.2 Function

For two given sets A and B a **rule** f which associates with **each** element of A, a **unique** element of B, is called a function from A to B. If f is a function from a set A to a set B then we denote the fact by **f: A → B.** For example the function f which associates the cube of a real number with a given real number x , can be written as $f(x) = x^3$
Suppose the value of x is 2 there f maps 2 to 8

Floor Function: Let x be a real number. The floor function denoted as $\lfloor x \rfloor$ maps each *real* number x to the *integer*, which is the greatest of all integers less than or equal to x.

For example: $\lfloor 3.5 \rfloor = 3$, $\lfloor -3.5 \rfloor = -4$, $\lfloor 8 \rfloor = 8$.

Ceiling Function: Let x be a real number. The ceiling function denoted as $\lceil x \rceil$ m*aps each* real *number x to the* integer*, which is the least of all integers greater than or equal to x.*

For example: $\lceil 3.5 \rceil = 4$, $\lceil -3.5 \rceil = -2$, $\lceil 8 \rceil = 8$

Next, we state a useful result, without proof.

For every real number x, **we have**

$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$.

2.2.3 Logarithms

Logarithms are important mathematical tools which are widely used in analysis of algorithms.

The following important properties of logarithms can be derived from the properties of exponents. However, we just state the properties without proof.

It general the logarithms of any number x is the power to which another number a, called the base, must be raised to produce x. Both a & x are positive numbers.

For n, a some important formulas related to logarithms are given below

(i)	$\log_a (bc)$	=	$\log_a b + \log_a c$
(ii)	$\log_a (b^n)$	=	$n \log_a b$
(iii)	$\log_b a$	=	$\log_a b$
(iv)	$\log_a (1/b)$	=	$-\log_b a$
(v)	$\log_a b$	=	$\frac{1}{\log_b a}$
(vi)	$a^{\log_b c}$	=	$c^{\log_b a}$

Modular Arithmetic/Mod Function

The modular function or mod function returns the remainder after a number (called dividend) is divided by another number called divisor. Many programming language has a similar function.

Definition

b mod n: if n is a given *positive* integer and b is *any* integer, then

b mod n = r where $0 \le r < n$ and $b = k * n + r$

In other words, r is obtained by subtracting multiples of n from b so that the remainder r lies between 0 and (n − 1).

For example: if b = 42 and n = 11 then

b mod n = 42 mod 11 = 9.

If b = − 42 and n = 11 then

b mod n = − 42 mod 11 = 2 ($\because -42 = (-4) \times 11 + 2$)

2.2.4 Theorem, Lemma & Corollary

These terms are often used when writing proofs in algorithms and mathematics but they are not identical
A **statement** is a sentence which has objective and logical meaning.

A **proposition** is a mathematical statement that is either true or false. For example, 5 is greater than 8 is a proposition , although not true. ′3′ a prime number is an another proposition, but true.

A **theorem** is a proposition which needs to be proven. Each of the examples of the arithmetic series could be stated as a theorem. For example, we could state the following theorem and then write a proof.
Theorem:

For all integers $n > 0$,
$1^2 + 2^2 + 3^2 + \ldots\ldots\ldots n^2 = \frac{n(n+1)(2n+1)}{6}$

Proof : The proof would be written here. In this case, it would be the induction proof.

Lemma – It can be defined as a subsidiary proposition employed to prove another theorem. When the proof of a theorem relies on the proof of some auxiliary propositions, lemmas are stated and proven concerning those propositions. These lemmas are then employed to prove the theorem. Lemmas are smaller results to be used in a theorem which is a bigger and more important result.

A **corollary** is a result from a theorem that doesn’t require too much proof to show. Corollaries are special cases of theorems.

2. 3 Some Mathematical Expectation

In average-case analysis of algorithms, we need the concept of **Mathematical expectation.** In order to understand the concept better, let us first consider an example.

Example 2.1: Suppose, the students of MCA, who completed all the courses in the year 2005, had the following distribution of marks.

Range of marks	**Percentage of students who scored in the range**
0% to 20%	08
20% to 40%	20
40% to 60%	57
60% to 80%	09
80% to 100%	06

If a student is picked up randomly from the set of students under consideration, what is the % of marks *expected* of such a student? After scanning the table given above, we *intuitively* expect the student to score around the 40% to 60% class, because, more than half of the students have scored marks in and around this class.

Assuming that marks within a class are uniformly scored by the students in the class, the above table may be approximated by the following more concise table:

% marks	**Percentage of students scoring the marks**
10*	08
30	20
50	57
70	09
90	06

As explained earlier, we *expect* a student picked up randomly, to score around 50% because more than half of the students have scored marks around 50%.

This *informal* idea of *expectation* may be formalized by giving to each percentage of marks, weight in proportion to the number of students scoring the particular percentage of marks in the above table.

Thus, we assign weight (8/100) to the score 10% (□□*8, out of 100 students, score on the average 10% marks*); (20/100) to the score 30% and so on.

Thus

Expected% = **of marks** $10 \times \frac{8}{100} + 30 \times \frac{20}{100} + 50 \times \frac{57}{100} + 70 \times \frac{9}{100} + 90 \times \frac{6}{100} = 47$

The final calculation of expected marks of 47 is roughly equal to our intuition of the expected marks, according to our intuition, to be around 50.

We generalize and formalize these ideas in the form of the following definition.

Mathematical Expectation

For a given set S of items, let to each item, one of the n values, say, v_1, $v_2,\ldots,v_n$, be associated. Let the probability of the occurrence of an item with value v_i be p_i. If an item is picked up at random, then its expected value E(v) is given by

$$E(v) = \sum_{i=1}^{n} p_i v_i = p_1.v_1 + p_2.v_2 + \cdots \ldots \ldots\ p_n.v_n$$

2. 4 The Principle of Induction

Induction plays an important role to many facets of data structure and algorithms. In general, all correctness opinions are based on induction principle.

Mathematical Induction is a method of writing a mathematical proof generally to establish that a given statement is true for all natural numbers. Initially we have to prove that the first statement in the infinite sequence of statements is true, and then proving that if any one statement in the infinite sequence of statement is true, then so is the next statement.

Therefore the method consists of the following three major steps:

1. **Induction base-** In this stage we verify/establish the correctness of the initial value(base case). It is the proof that the statement is true for n = 1 or some other starting value.
2. Induction Hypothesis- it is the assumption that the statement is true for any value of n where $n \geq 1$
3. Induction Step- In this stage we make a proof that if the statement is true for n , it must be true for n+1

Example 1: Write a proof that the sum of the first n positive integers is $\frac{\mathbf{n(n+1)}}{\mathbf{2}}$, that is

$$\mathbf{1 + 2 + \cdots \ldots + n = \frac{n(n+1)}{2}}.$$

Proof: (Through Induction Method)

Base Step): We must show that the given equation is true for n=1

i.e. $1 = \frac{1(+1)}{2} = 1 \Longrightarrow$ this is true.

Hence we proved that the first statement is true for n = 1

Induction Hypothesis: Let us assume that the given equation is true for any value of n (n ≥ 1)

that is $1 + 2 + \cdots ... + n = \frac{n(n+1)}{2}$;

Induction Step: Now we have to prove that it is true for (n+1).

Consider

$$1 + 2 + 3 + \cdots ... ... + n + (n + 1) = \frac{(n+1)[(n+1)+1]}{2}$$

Let us rewrite the above equation in the following way:

$$\mathbf{1 + 2 + 3} + \cdots ... ... + \mathbf{n} + (n + 1) = \frac{\mathbf{n(n+1)}}{\mathbf{2}} + n+1$$

$$= \frac{n(n+1)2(n+1)}{2}$$

$$= \frac{(n+1)(n+2)}{2}$$

$$= \frac{(n+1)[(n+1)+1]}{2}$$

Therefore, if the hypothesis is true for n, it must be true for n+1 which we proved in induction step.

Check your progress-1

Question-1: Prove that for all positive integers n,

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Question2: Prove that for all nonnegative integers n,

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

2. 5 Concept of Efficiency of An Algorithm

In order to understand the complexity/efficiency of an algorithm, it is very important to understand the notion of the **size of an instance** of the problem under consideration and the role of size in determining complexity of the solution. For example finding the product of two 2 × 2 matrices will take much less time than the time taken by the same algorithm for multiplying say two 100 × 100 matrices. This explains intuitively *the notion of the size of an instance of a problem* and also the role of size in determining the (*time*) complexity of an algorithm. **If the *size* (to be later considered formally) of general *instance is n* then time complexity of the algorithm solving the *problem* (not just the instance) under consideration is *some function of n.***

In view of the above explanation, the notion of *size* of an instance of a problem plays an important role in determining the complexity of an algorithm for solving the problem under consideration. However, it is difficult to *define precisely* the concept of size in general, for all problems that may be attempted for algorithmic solutions. In some problems ***number of bits is*** *required in representing the size of an instance.* However, for all types of problems, this does not serve properly the purpose for which the notion of size is taken into consideration. Hence different measures of size of an instance of a problem are used for different types of problems. Let us take two examples:

(i) In sorting and searching problems, the *number of elements*, which are to be sorted or are considered for searching, is taken *as the size of the instance* of the problem of sorting/searching.
(ii) In the case of solving polynomial equations or while dealing with the algebra of polynomials, the *degrees* of polynomial instances, may be taken as the sizes of the corresponding instances.

To measure the efficiency of an algorithm, we will consider the theoretical approach and follow the following steps:

- Calculation of time complexity of an algorithm- *Mathematically* determine the time needed by the algorithm, for a *general instance* of size, say, n of the problem under consideration. In this approach, generally, each of the basic instructions like *assignment*, *read* and *write*, arithmetic operations, comparison operations are assigned some constant number of (basic) units of time for execution. Time for looping statements will depend upon the number of times the loop executes. Adding basic units of time of all the instructions of an algorithm will give the total amounts of time of the algorithm
- The approach does not depend on the programming language in which the algorithm is coded and on how it is coded in the language as well as the computer system used for executing (a programmed version of) the algorithm. But different computers have different execution speeds. However, the speed of one computer is generally some constant multiple of the speed of the other
- In stead of applying the algorithm to many different-sized instances, the approach can be applied for a *general size say n* of an arbitrary instance of the problem but the size n may *be arbitrarily large* under consideration.

An ***important consequence of the above discussion*** *is that if the time taken by one machine in executing a solution of a problem is a polynomial (or exponential) function in the size of the problem, then time taken by every machine is a polynomial (or exponential) function respectively, in the size of the problem.*

In the next section we will examine the asymptotic approach to analyze the efficiency of algorithms

2. 6 Asymptotic Analysis and Notations

Asymptotic analysis" is a more formal method for analyzing algorithmic efficiency. It is a mathematical tool to analyze the time and space complexity of an algorithm as a function of input size. For example, when analyzing the time complexity of any sorting algorithm such as Bubble sort, Insertion sort and Selection sort in the worst case scenario , we will be concerned with how long it takes as a function of the length of the input list. For example, we say the time complexity of standard sorting algorithms in the worst case takes T(n) = n^2 where n is a size of the list . In contrast, Merge sort takes time $T(n) = n*log_2(n)$.

The **asymptotic** behavior of a function *f(n)* (such as *f(n)=c*n* or $f(n)=c*n^2$, etc.) refers to the growth of *f(n)* as *n* gets very large. Small values of *n* are not considered. The main concern in asymptotic analysis of a function is in estimating how slow the program will be on large inputs. One should always remember: the slower the asymptotic growth rate, the better the algorithm. The Merge sort algorithm is better than the standard sorting algorithms. Binary search algorithm is better than the linear searching algothm. A linear algorithm is always asymptotically better than a quadratic one .. Remember to think a very large input size when working with asymptotic rates of growth. If the relative behaviors of two functions for smaller values conflict with the relative behaviors for larger values ,then we may ignore the conflicting behaviors for smaller values.

For example, let us consider the time complexities of two solutions of a problem having input size n as given below:

T_1(n) = 1000 n^2

T_2(n) = 5n^4

Despite the fact T_1(n) ≥ T_2(n) for n ≤ 14, we would still prefer the solution as T_1(n) as the time complexity because

T_1(n) ≤ T_2(n) for all n ≥ 15

Some common orders of growth seen often in complexity analysis are

O(1)	constant
O(log n)	logarithmic
O(n)	linear
O(n log n)	"n log n"
$O(n^2)$	quadratic
$O(n^3)$	Cubic
O(n^k)	polynomial
O(2^n)	exponential

2.6.1 Worst Case and Average Case Analysis

Consider a linear search algorithm. The worst case of the algorithm is when the element to be searched for is either not in the list or located at the end of the list. In this case the algorithm runs for the longest possible time. It will search the entire list. If an algorithm runs in time *T(n)*, we mean that *T(n)* is an upper bound on the running time that holds for all inputs of size *n*. This is called *worst-case analysis*.
A popular alternative to worst-case analysis is *average-case analysis* which provides average amount of time to solve a problem. Here we try to calculate the expected time spent on a randomly chosen input. This kind of analysis is generally more difficult compared to worst case analysis. Because it involves

probabilistic arguments and often requires assumptions about the distribution of inputs that may not be very easy to justify. But sometimes it can be more useful compared to the worst-case analysis of an algorithm. A Quicksort algorithm, whose worst-case running time on an input sequence of length n is proportional to n^2 but whose expected running time is proportional to $n \log n$.

2.6.2 Drawbacks of Asymptotic Analysis

- Let us consider two standard sorting algorithms : The first takes 1000 n^2 and the second takes 10 n^2 time in the worst case respectively on a machine. Both of these algorithms are asymptotically same (order of growth isn^2). Since we ignore constants in asymptotic analysis, it is difficult to judge which one is more suitable.
- Worst case versus average performance
  If an algorithm *A* has better worst case performance than the algorithm *B*, but the average performance of *B* given the expected input is better, then *B* could be a better choice than *A*.

2.7 Asymptotic Notations

There are mainly three asymptotic notations if we do not want to get involved with constant coefficients and less significant terms. These are

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

2.7.1 `Big-O notation: Upper Bounds (maximum number of steps to solve a problem).Big O is used to represent the upper bound or a worst case of an algorithm since it bounds the growth of the running time from above for large value of input sizes. It notifies that a particular procedure will never go beyond a specific time for every input n. One important advantage of big-O notation is that it makes algorithms much easier to analyze, since we can conveniently ignore low-order terms.

Formally, big-Oh notation can be defined as follows-

Let $f(n)$ and $g(n)$ are two positive functions , each from the set of natural numbers (domain) to the positive real numbers.

We say that the function $f(n) = O(g(n))$ [read as "f of n is big "Oh" of g of n"], if there exist two positive constants c and n_0 such that

$$f(n) \leq C.g(n) : \ \forall \ n \geq n_0$$

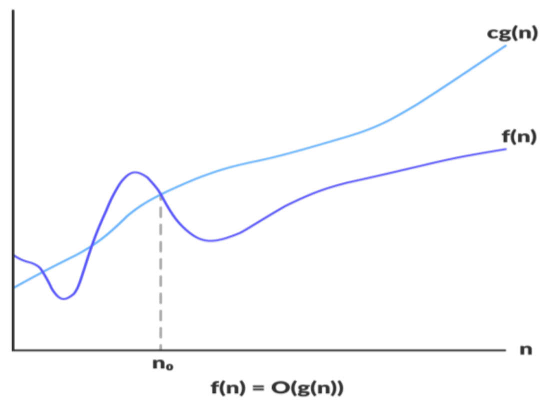


Figure f(n) = O(g(n))

When a running time of f(n) is O(g(n)), it means the running time of a function is bounded from above for input size n by c.g(n)

Consider the following examples

1. Verify the complexity of $3n^2 + 4n - 2$ is $O(n^2)$?

In this case $f(n) = 3n^2 + 4n - 2$ and $g(n) = n^2$

The above function is still a quadratic algorithm and can be written as:

$3n^2 + 4n - 2 <= 3n^2 + 4n^2 - 2n^2$

$<= (3 + 4 - 2)\, n^2$

$= O(n^2)$

One important advantage of big-O notation is that it makes algorithms much easier to analyze, since we can conveniently ignore low-order terms and constants

Now to find out what values of c and n_0 , so that

$3n^2 + 4n - 2 <= cn^2$ for all $n >= n_0$.

If n_0 is 1, then c must be greater than or equal to $3 + 4 - 2 <= c$, *i.e., 6. So, above function can now be written as-*

$3n^2 + 4n - 2 <= 6n^2$ for all $n >= 1$

So, we can say that $3n^2 + 4n - 2 = O(n^2)$.

2. Show $n^3 != O(n^{2)\cdot}$

First select c and n_0 so that:

$n^3 <= cn^2$ for each $n >= n_0$

Now both sides are divided by n^2

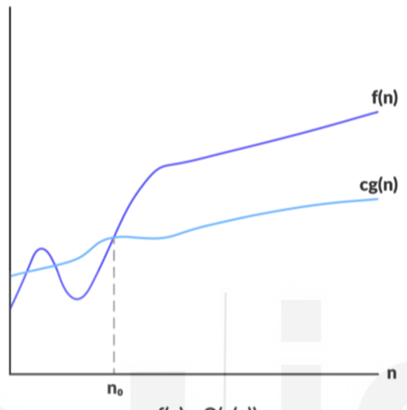
$n <= c$ for all $n >= n_0$

However, it will never be possible, thus the statement that $n^3 = O(n^{2)}$ must be incorrect.

2.7.2 Big-Omega(Ω) Notation

Big Omega (Ω) describes the asymptotic **lower bound** of an algorithm whereas a big Oh(O)notation represents an upper bound of an algorithm. Generally we say that an algorithm takes at least this amount of time without mentioning the upper bound. In such case, big-(Ω) notation is applied. Let's define it more formally:

f(n) = Ω(g(n)) if and only if there exists some constants C and n_0 such that f(n) ≥ C.g(n) : ∀ n ≥ n_0 . The following graph illustrates the growth of f(n) = Ω(g(n))



f(n) = Ω(g(n))

As shown in the above graph f(n) is bounded from below by C.g(n). Note that for all values of f(n) always lies on or above g(n).
If f(n) is Ω(g(n)) which means that the growth of f(n) is asymptotically no slower than g(n) no matter what value of n is provided.

Example 1

Example1.1: For the function defined by

$f(n) = 2n^3 + 3n^2 + 1$ and
clearly this equation (1) is satisfied for $C = 1$ and for all $n \geq 1$

$g(n) = 2n^2 + 3$: show that

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

To show that $(n) = \Omega(g(n))$; we have to show that

$$f(n) \geq C.g(n) \; \forall n \geq n_0$$

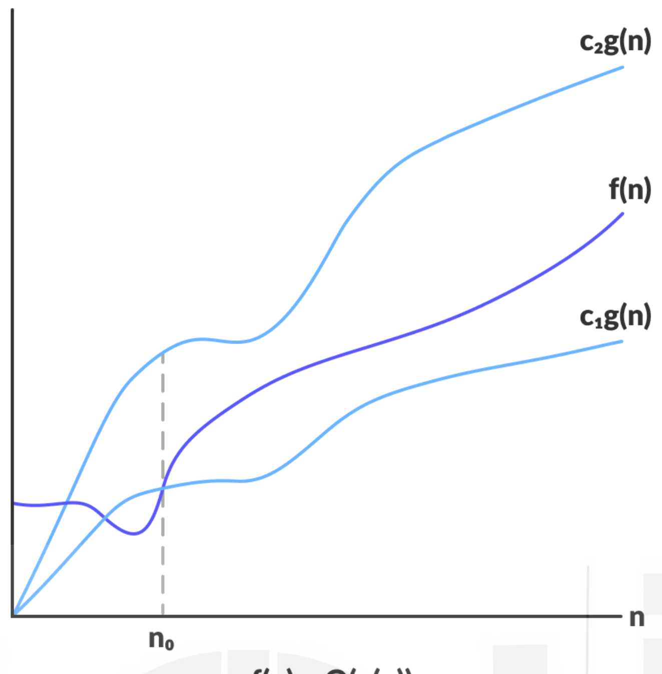
$$\Rightarrow 2n^3 + 3n^2 + 1 \geq C(2n^2 + 3) \quad \ldots..(1)$$

Since we have found the required constant C and $n_0 = 1$.

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

2.7.3 Θ (Theta) notation: Tight Bounds

In case the running time of an algorithm is Θ(n), it means that once n gets large enough, the running time is minimum c1·n, and maximum c2·n, where c1 and c2 are constants. It provides both upper and lower bounds of an algorithm. The following figure illustrates the function f(n) = Θ(g(n). As shown in the figure the value of f(n) lies between c1(g(n)) and c2(g(n))for sufficiently large value of n.



f(n) = Θ(g(n))

Now let us define the theta notation: for a given function g(n) and constants C_1, C_2 and n_0 where $n_0>0$, $C_1>0$, and $C_2>0$, $\Theta(g(n))$ can be denoted as a set of functions such that the following condition is satisfied:
$0 <= C_1g(n) <= f(n) <= C_2g(n)$ for all $n >= n_0$
The above *inequalities represent two conditions to be satisfied simultaneously viz., $C_1\, g(x) \leq f(x)$ and $f(x) \leq C_2\, g(x)$)*

The following theorem which relates the three functions O, Ω, Θ does not have proof:

Theorem: For any two functions f(x) and g(x), f(x) = Θ (g(x)) if and only if *f(x) = O (g(x)) and f(x) = Ω (g(x)).*

if f(n) is Θ(g(n)) this means that the growth of f(n) is asymptotically at the

same rate as g(n) or we can say the growth f(n) is not asymptotically

slower or faster than g(n) no matter what value of n is provided

2.7.4 Some Useful Theorems for O, Ω, Θ

The following theorems are quite useful when you are dealing (or solving problems) with *O*, *Ω* and *Θ*

Theorem 1: If $f(n) = a_mn^m + a_{m-1}n^{m-1} + \cdots \dots \dots \dots \dots + a_1n + a_0$

Where $a_m \neq 0\ and\ a_i \in R, then\ f(n) = O(n^m)$

Proof: $f(n) = a_mn^m + a_{m-1}n^{m-1} + \cdots \dots \dots \dots + a_1n + a_0$

$$= \sum_{i=0}^{m} a_k n^k$$

$$f(n) \le \sum_{i=0}^{m} |a_k| n^k$$

$$\le n^m \sum_{i=0}^{m} |a_k| n^{k-m} \le n^m \sum_{i=0}^{m} |a_k| \text{for } n \ge 1$$

Let us assume $|a_m| + |a_{m-1}| + \cdots \ldots \ldots \ldots + |a_1| + |a_0| = c$

Then $f(n) \le \ cn^m \Rightarrow f(n) = O(n^m)$.

Theorem 2: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots \ldots \ldots \ldots \ldots + a_1 n + a_0$

Where $a_m \neq and\ a_i \in R, then\ f(n) = \Omega\ (n^m)$.

Proof: $f(n) = \ a_m n^m + \cdots \ldots \ldots \ldots . a_1 n + a_0$

Since $f(n) \ge \ cn^m\ for\ all\ n \ge 1 \ \Rightarrow f(n) = \ \Omega(n^m)$

Theorem 3: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots \ldots \ldots \ldots \ldots a_1 n + a_0$

Where $a_m \neq 0\ and\ a_i \in R, then\ f(n) = 0\ (n^m)$.

Proof: From Theorem 1 and Theorem 2,

$f(n) = O\ (n^m) \ldots \ldots \ldots . (1)$

$f(n) = \Omega\ (n^m) \ldots \ldots \ldots . . (2)$

From (1) and (2) we can say that $f(n) = \ \Theta(n^m)$

Example 1: By applying theorem, find out the O-notation, Ω-notation and Θ-notation for the following functions.

(i) $f(n) = \ 5n^3 + 6n^2 + 1$
(ii) $f(n) = \ 7n^2 + 2n + 3$

Solution:

(i) Here *The degree of a polynomial* f(n) is, m = 3, So by Theorem 1, 2 and 3:
$f(n) = O(n^3), f(n) = \Omega\ (n^3)$and $f(n) = \Theta(n^3)$,

(ii) The degree of a polynomial $f(n) is\ , m = 2, So\ by\ theorem\ 1,2\ and\ 3$:
$f(n) = O(n^2), f(n) = \Omega\ (n^2)$and $f(n) = \Theta(n^2)$,

Check your progress-2

Let f(n) and g(n) be two asymptotically positive functions. Prove or disprove the following (using the basic definition of O, Ω andΘ):

a) $4n^2 + 7n + 12 = O(n^2)$
b) $logn + \log(logn) = O(logn)$

c) $3n^2 + 7n - 5 = \Theta(n^2)$

d) $2^{n+1} = O(2^n)$

e) $2^{2n} = O(2^n)$

f) $f(n) = O\left(g(n)\right) implies\ g(n) = O\left(f(n)\right)$

g) max{$f(n)$, $g(n)$} = $\Theta\left(f(n) + g(n)\right)$

h) $f(n) = O\left(g(n)\right) implies\ 2^{f(n)} = O\left(2^{g(n)}\right)$

i) $f(n) + g(n) = \Theta(\min\left(f(n), g(n)\right)$

j) $33n^3 + 4n^2 = \Omega(n^2)$

(

2.8 Summary

Before solving any problem, an algorithm is designed. An algorithm is a definite, step-by-step procedure for performing some tasks.. To analyze the algorithms efficiency, asymptotic notations are generally used. There are three popular asymptotic notations namely, big O, big Ω, and big Θ. Big O notation is mostly used as it measures the upper bound of complexity. On the contrary, big omega measures the lower bound of time complexity and big theta is used to measure the average time of any algorithm.

2.9 Solution of Check Your Progress

Check Your Progress1

Question-1: Prove that for all positive integers n,

$$1^2 + 2^2 + \cdots + \mathrm{n}^2 = \frac{\mathrm{n}(\mathrm{n}+1)(2\mathrm{n}+1)}{6}.$$

Proof: (Base Step):

Consider n=1, then

$$1^2 = \frac{1(1+1)(2+1)}{6} = \frac{1.2.3}{6} = 1$$

Hence for n=1 it is true.

Induction Hypothesis: Assume the above statement is true for 'n' i.e.

$$1^2 + 2^2 + 3^2 + \cdots \ldots \ldots + \mathrm{n}^2 = \frac{\mathrm{n}(\mathrm{n}+1)(2\mathrm{n}+1)}{6}.$$

Induction Step:

Now we need to show that

$$1^2 + 2^2 + \cdots + (n+1)^2 = \frac{(n+1)[(n+1)+1][2(n+1)+1]}{6}$$

To that end,

$$1^2 + 2^2 + \cdots + (n+1)^2 = \boldsymbol{1^2 + 2^2 + \cdots + n^2} + (n+1)^2$$
$$= \boldsymbol{\frac{n(n+1)(2n+1)}{6} + (n+1)^2}$$

$$= \frac{n(n+1)(2n+1)+6(n+1)^2}{6}$$
$$= \frac{(n+1)(2n^2+n+6n+6)}{6}$$
$$= \frac{(n+1)(2n^2+7n+6)}{6}$$
$$= \frac{(n+1)(n+2)(2n+3)}{6}$$
$$= \frac{(n+1)[(n+1)+1][2(n+1)+1]}{6}$$

Questioin2: Prove that for all nonnegative integers n,

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

We show, for all nonnegative integers n, that

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

In summation notation, this equality can be defined as

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Induction base: For n = 0,

$$2^0 = 1 = 2^{0+1} - 1.$$

Induction hypothesis: Assume, for an arbitrary nonnegative integer n, that

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

Induction step: We need to show that

$$2^0 + 2^1 + 2^2 + \cdots + 2^{n+1} = 2^{(n+1)+1} - 1.$$

To that end,

$$\begin{aligned} 2^0 + 2^1 + 2^2 + \cdots + 2^{n+1} &= \mathbf{2^0 + 2^1 + 2^2 + \cdots + 2^n} + 2^{n+1} \\ &= \mathbf{2^{n+1} - 1} + 2^{n+1} \\ &= 2(2^{n+1}) - 1 \\ &= 2^{(n+1)+1} - 1. \end{aligned}$$

Check Your Progress2

a) $4n^2 + 7n + 12 = O(n^2)$

$(4n^2 + 7n + 12) \le c, n^2 \ldots\ldots\ldots (1)$

for c = 5 and n ≤9; the above inequality (1) is satisfied.

Hence $4n^2 + 7n + 12 = O(n^2)$.

b) $logn + \log(logn) = O(logn)$

By using basic definition of Big –"oh" Notation:

$logn + \log(logn) \le C.\log n \ldots\ldots\ldots\ldots (1)$

For C = 2 and $n_0 = 2, we\ have$

$log_2 2 + \log(log_2 2) \le 2.log_2 2$

⇒ 1+ log(1) ≤ 2

⇒ 1+0 ≤ 2

⇒ Satisfied for c = 2 and $n_0 = 2$

$\Rightarrow$ *log n* +log(*logn*) = *O* (*logn*)

c) $3n^2 + 7n - 5 = \Theta(n^2)$

$3n^2 + 7n - 5 = \Theta(n^2);$

To show this, we have to show:

$C_1.n^2 \le 3n^2 + 7n - 5 \le C_2.n^2 \ldots\ldots\ldots (*)$

(i) L.H.S inequality

$C_1.n^2 \le 3n^2 + 7n - 5 \ldots\ldots (1)$

This is satisfied for $C_1 = 1$and n ≥ 2

(ii) R.H.S inequality

$3n^2 + 7n - 5 \le C_2.n^2 \ldots\ldots (2)$

This is satisfied for $C_1 = 1 \; and \; n \ge 1$

$\Rightarrow$ inequality (*) is simultaneously satisfied for

$C_1 = 1, C_2 = 10 \; and \; n \ge 2$

d) $2^{n+1} = O(2^n)$

$2^{n+1} \le C.2^n \implies 2^{n+1} \le 2.2^n$

e) $2^{2n} = O(2^n)$

$2^{2n} = \le C.2^n$

$\Rightarrow$ $4^n \le 2.2^n$……(1)

No value of C and n_0 Satisfied this in equality (1)

$\Rightarrow$ $2^{2n} \ne O(2^n)$.

f) $f(n) = O\left(g(n)\right) implies \; g(n) = O(f(n))$

No; *f*(*n*) = *O*(g(n)) does not implies *g*(*n*) =*O*(*f*(*n*)

Clearly n = *O*(n^2), but $n^2 \ne O(n)$

g) max{*f*(*n*), *g*(*n*)} = $\Theta\left(f(n) + g(n)\right)$

To prove this, we have to show that

$$C_1.(f(n) + g(n) \le \max\{f(n), g(n)\}$$
$$\le C_2(f(n)+g(n) \ldots\ldots\ldots\ldots(*)$$

1) L.H.S inequality:

$C_1.(f(n) + g(n) \le \max\{f(n), g(n)\} \ldots\ldots\ldots\ldots (1)$

Let h(n) =max {f(n), g(n)) = $\begin{cases} f(n) & if \;\; f(n) > g(n) \\ g(n) & if \;\; g(n) > f(n) \end{cases}$

$\therefore$ $C_1.(f(n) + g(n) \le f(n) \ldots\ldots\ldots.(1)$

[Assume max {*f*(*n*), *g*(*n*)} = *f*(*n*)]

$for \; C_1 = \dfrac{1}{2} and \; ; n \ge 1,$ this inequality (1) is satisfied:

$$Since\ \frac{1}{2}(f(n) + g(n) \le f(n)$$

$\Rightarrow \quad f(n) + g(n) \le 2f(n)$

$\Rightarrow \quad f(n) + g(n) \le f(n) + f(n) \quad [\because f(n) > g(n)]$

$\Rightarrow$ Satisfied for $C_1 = \frac{1}{2}\ and\ n \ge 1$

2) R.H.S inequality

max{*f*(*n*), *g*(*n*)} ≤ C_1 . (*f*(*n*) +*g*(*n*) ………(2)

This inequality (2) is satisfied for $C_2 = 1$ and n ≥ 1

$\Rightarrow$ inequality (*) is simultaneously satisfied for

$C_1 = \frac{1}{2}, C_2 = 1\ and\ n \ge 1$

Remark: Let *f*(*n*) = n and *g*(*n*) =n^2;

then max$\{n,n^2\} = \Theta\ (n + n^2)$

$\Rightarrow n^2 = \Theta\ (n^2)$; which is TRUE (by definition of Θ)

h) $f(n) = O(g(n)) implies\ 2^{f(n)} = O(2^{g(n)})$

$No; f(n) = O\ (g(n) does\ not\ implies\ 2^{f(n)} = O(2^{g(n)})$;

we can prove this by taking a counter Example ;

Let $f(n) = 2n\ and\ g(n) = n$, we have

$2^{2n} = O(2^n)$; which is not TRUE $[since\ 2^{2n} = 4^n \ne O(2^n)]$.

i) $f(n) + g(n) = \Theta(\min(f(n), g(n))$

No, *f*(*n*)+g(n) $\ne \Theta(\min\{f(n), g(n)\})$

We can prove this by taking counter example.

Let $f(n) = 2n\ and\ g(n) = n^2$, then $(n + n^2) \ne \Theta(n)$

j) $33n^3 + 4n^2 = \Omega(n^2)$

$(33n^3 + 4n^2) \ge C.n$;

There is no positive integer for C and n_0

which satisfy this inequality. Hence $(33n^3 + 4n^2) \ne C.n^4$.