UNIT 2 SOME PRE-REQUISITES AND ASYMPTOTIC BOUNDS

Structure **Page Nos.**

2.0 INTRODUCTION

We have already mentioned that there may be more than one algorithms, that solve a given problem. In Section 3.3, we shall discuss eight algorithms to sort a given list of numbers, each algorithm having its own merits and demerits. Analysis of algorithms, the basics of which we study in Unit 3, is an essential tool for making well-informed decision in order to choose the most suitable algorithm, out of the available ones if any, for the problem or application under consideration.

A number of mathematical and statistical tools, techniques and notations form an essential part of the baggage for the analysis of algorithms. We discuss some of these tools and techniques and introduce some notations in Section 2.2. However, for detailed discussion of some of these topics, one should refer to the course material of *MCS-013*.

Also, in this unit, we will study a number of well-known *approximation functions*. These approximation functions which calculate approximate values of quantities under consideration, prove quite useful in many situations, where some of the involved quantities are calculated just for comparison with each other. And the correct result of comparisons of the quantities can be obtained even with approximate values of the involved quantities. In such situations, the advantage is that the approximate values may be calculated much more efficiently than can the actual values.

The understanding of the theory of a routine may be greatly aided by providing, at the time of construction one or two statements concerning the state of the machine at well chose points…In the extreme form of the theoretical method a watertight mathematical proof is provided for the assertions. In the extreme form of the experimental method the routine is tried out one the machine with a variety of initial conditions and is pronounced fit if the assertions hold in each case. Both methods have their weaknesses.

A.M. Turing
Ferranti Mark 1
Programming Manual (1950)

2.1 OBJECTIVES

After going through this Unit, you should be able to:

- use a number of mathematical notations e.g., Σ, $\prod$, $\lfloor\ \rfloor$, $\lceil\ \rceil$, mod, log, e etc.

- define and use a number of concepts like function, 1-1 function, onto function, monotonic function, floor and ceiling functions, mod function, exponentiation, logarithm functions etc
- define and use Mathematical Expectation
- Use of Principle of Mathematical Induction in establishing truth of infinitely many statements and
- define and use the five asymptotic function viz.,

(i) O: (O (n^2) is pronounced as 'big-oh of n^2' or sometimes just as oh of n^2)

(ii) Ω: (Ω (n^2) is pronounced as 'big-omega of n^2 or sometimes just as omega of n^2')

(iii) Θ: (Θ (n^2) is pronounced as 'theta of n^2')

(iv) o: (o (n^2) is pronounced as 'little-oh of n^2')

(v) ω: (ω (n^2) is pronounced as 'little-omega of n^2').

2.2 SOME USEFUL MATHEMATICAL FUNCTIONS & NOTATIONS

Let us start with some mathematical definitions.

2.2.1 Functions & Notations

Just to put the subject matter in proper context, we recall the following notations and definitions.

Unless mentioned otherwise, we use the letters N, I and R in the following sense:

N = {1, 2, 3, …}
I = {…, − 2, −, 0, 1, 2, ….}
R = set of Real numbers.

Notation 2.2.1.1: If $a_1, a_2 \ldots a_n$ are n real variables/numbers then

(i) Summation:

The expression
$a_1 + a_2 + \ldots + a_i + \ldots + a_n$
may be denoted in shorthand as

$$\sum_{i=1}^{n} a_i$$

(ii) Product

The expression
$a_1 \times a_2 \times \ldots \times a_i \times \ldots \times a_n$
may be denoted in shorthand as

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

Definition 2.2.1.2:

Function:

For two given sets A and B (*which need not be distinct, i.e., A may be the same as 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: $A \rightarrow B$.** Also, for $x \in A$, f(x) is called **image** of x in B. Then, A is called the **domain** of f and B is called the **Codomain** of f.

Example 2.2.1.3:

Let f: $I \rightarrow I$ be defined such that

$f(x) = x^2$ for all $x \in I$

Then

f maps	-4	to	16
f maps	0	to	0
f map	5	to	25

Remark 2.2.1.4:

We may note the following:

(i) if f: $x \rightarrow y$ is a function, then there may be more than one elements, say x_1 and x_2 such that

$$f(x_1) = f(x_2)$$

For example, in the Example 2.2.1.3

$$f(2) = f(-2) = 4$$

By putting restriction that $f(x) \neq f(y)$ if $x \neq y$, we get special functions, called 1-1 or injective functions and shall be defined soon.

(ii) Though for each element $x \in X$, there must be at least one element $y \in Y$ s.t $f(x) = y$. However, it is not necessary that for each element $y \in Y$, there must be an element $x \in X$ such that $f(x) = y$. For example, for $y = -3 \in Y$ there is no $x \in X$ s.t $f(x) = x^2 = -3$.

By putting the restriction on a function f, that for each $y \in Y$, there must be *at least one* element x of X s.t f(x) = y, we get special functions called onto or surjective functions and shall be defined soon.

Next, we discuss some important functions.

Definition 2.2.1.5:

1-1 **or Injective Function:** A function f: $A \rightarrow B$ is said **to** 1-1[*] **or injective** if for $x, y \in A$, if $f(x) = f(y)$ then $x = y$

We have already seen that the function defined in Example 2.2.1.3 is **not** 1-1. However, by changing the domain, through defined by the same rule, f becomes a 1-1 function.

Example 2.2.1.2:

In this particular case, if we change the domain from I to N = {1,2,3…} then we can easily check that function

[*] Some authors write 1-to-1 in stead of 1-1. However, other authors call a function 1-to-1 if f is both 1-1 and onto (to be defined 0 in a short while).

$f : N \to I$ defined as
$f(x) = x^2$, for all $x \in N$,

is 1-1.

Because, in this case, for each $x \in N$ its negative $- x \notin N$. Hence for $f(x) = f(y)$ implies $x = y$. For example, If $f(x) = 4$ then there is **only** one value of x, viz, $x = 2$ s.t $f(2) = 4$.

Definition 2.2.1.7:

Onto/Surjective function: A function $f: X \to Y$ is said to **onto, or surjective** if to **every element of Y,** the codomain of f, there is an element $x \in X$ s.t $f(x) = y$.

We have already seen that the function defined in Example 2.2.1.3 is **not onto**.

However, in this case either, by changing the codomain Y or changing the rule, (or both) we can make f as Onto.

Example 2.2.1.8: (*Changing the domain*)

Let $X = I = \{\ldots - 3, - 2, -1, 0, 1, 2, 3, \ldots\}$, but, we change Y as
$Y = \{0, 1, 4, 9, \ldots\} = \{y \mid y = n^2 \text{ for } n \in X\}$
then it can be seen that

$f: X\to Y$ defined by
$f(x) = x^2$ for all $x \in X$ is Onto

Example 2.2.1.9: (***Changing the rule***)

Here, we change the rule so that $X = Y = \{\ldots - 3, -2, -1, 0, 1,2, 3\ldots\}$
But $f: X \to Y$ is defined as
$F(x) = x + 3$ for $x \in X$.

Then we apply the definition to show that f is onto.

If $y \in Y$, then, by definition, for f to be onto, there must exist an $x \in X$ such that $f(x) = y$. So the problem is to find out $x \in X$ s.t $f(x) = y$ (*the given element of the codomain y*).

Let us *assume* that $x \in X$ exists such that

$f(x) = y$
i.e., $x + 3 = y$
i.e., $x = y - 3$

But, as y is given, x is known through the above equation. Hence f is onto.

Definition 2.2.1.10:

Monotonic Functions: For the definition of monotonic functions, we consider only functions

$f: R \to R$

where, R is the set of real numbers[*].

[*] Monotonic functions
$f : X \to Y$,
may be defined even when each of X and Y, in stead of being R, may be any **ordered** sets. But, such general definition is not required for our purpose.

A function f: $R \to R$ is said to be **monotonically increasing** if for $x, y \in R$ and $x \le y$ we have $f(x) \le f(y)$.

In other words, as x increases, the value of its image f(x) also increases for a monotonically increasing function.

Further, f is said to be ***strictly monotonically* increasing**, if $x < y$ then $f(x) < f(y)$

Example 2.2.1.11:

Let f: $R \to R$ be defined as $f(x) = x + 3$, for $x \in R$

Then, for $x_1, x_2 \in R$, the domain, if $x_1 \ge x_2$ then $x_1 + 3 \ge x_2 + 3$, (*by using monotone property of addition*), which implies $f(x_1) \ge f(x_2)$. Hence, f is monotonically increasing.

*We will discuss after a short while, useful functions called **Floor and Ceiling functions** which are monotonic but **not strictly** monotonic.*

A function f: $R \to R$ is said to be **monotonically decreasing**, if, for $x, y \in R$ and $x \le y$ then $f(x) \ge f(y)$.
In other words, as x increases, value of its image decreases.

Further, f is said to be *strictly monotonically decreasing*, if $x < y$ then $f(x) > f(y)$.

Example 2.2.1.12:

Let f: $R \to R$ be defined as
$F(x) = -x + 3$
if $x_1 \ge x_2$ then $-x_1 \le -x_2$ implying $-x_1+3 \le -x_2 + 3$,
which further implies $f(x_1) \le f(x_2)$
Hence, f is monotonically decreasing.

Next, we define Floor and Ceiling functions which map every *real* number to an *integer*.

Definition 2.2.1.13:

Floor Function: maps each *real* number x to the *integer*, which is the greatest of all integers less than or equal to x. **Then the image of x is denoted by $\lfloor x \rfloor$.**

Instead of $\lfloor x \rfloor$, the notation [x] is also used.

For example: $\lfloor 2.5 \rfloor = 2$, $\lfloor -2.5 \rfloor = -3$, $\lfloor 6 \rfloor = 6$.

Definition 2.2.1.14:

Ceiling Function: *maps each* real *number x to the* integer, *which is the least of all integers greater than or equal to x.* ***Then the image of x is denoted by $\lceil x \rceil$.***

For example: $\lceil 2.5 \rceil = 3$, $\lceil -2.5 \rceil = -2$, $\lceil 6 \rceil = 6$

Next, we state a useful result, without proof.

Result 2.1.1.10: For every real number x, **we have**

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

Example 2.2.1.15:

Each of the floor function and ceiling function is a *monotonically increasing* function but not **strictly** *monotonically increasing function*. Because, for real numbers x and y, if $x \le y$ then $y = x + k$ for some $k \ge 0$.

$\lfloor y \rfloor = \lfloor x + k \rfloor$ = integral part of $(x + k) \ge$ integral part of $x = \lfloor x \rfloor$

Similarly

$\lceil y \rceil = \lceil x + k \rceil$ = least integer greater than or equal to $x + k \ge$ least integer greater than or equal to $x = \lceil x \rceil$.

But, each of floor and ceiling function is **not strictly** increasing, because

$\lfloor 2.5 \rfloor = \lfloor 2.7 \rfloor = \lfloor 2.9 \rfloor = 2$

and

$\lceil 2.5 \rceil = \lceil 2.7 \rceil = \lceil 2.9 \rceil = 3$

2.2.2 Modular Arithmetic/Mod Function

We have been implicitly performing modular arithmetic in the following situations:

(i) If, we are following 12-hour clock, (*which usually we do*) and if it 11 O'clock now then after 3 hours, it will be 2 O'clock and not 14 O'clock (*whenever the number of o'clock exceeds 12, we subtract n = 12 from the number*)

(ii) If, it is 5th day (*i.e., Friday*) of a week, after 4 days, it will be 2nd day (*i.e., Tuesday*) and not 9th day, *of course of another,* week (*whenever the number of the day exceeds 7, we subtract n = 7 from the number, we are taking here Sunday as 7th day, in stead of 0th day*)

(iii) If, it is 6th month (*i.e., June*) of a year, then after 8 months, it will be 2nd month (*i.e., February*) of, *of course another,* year (*whenever, the number of the month exceeds 12, we subtract n = 12*)

In general, with minor modification, we have

Definition 2.2.2.1:

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

$b \bmod n = r$ 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
$b \bmod n = 42 \bmod 11 = 9$.

If $b = -42$ and $n = 11$ then
$b \bmod n = -42 \bmod 11 = 2$ (Θ $-42 = (-4) \times 11 + 2$)

Mod function can also be expressed in terms of the floor function as follows:

$b \pmod n = b - \lfloor b/n \rfloor \times n$

Definition 2.2.2.2:

Factorial: For N = {1,2,3,…}, the factorial function

factorial: $N \cup \{0\} \rightarrow N \cup \{0\}$

given by

factorial (n) = n $\times$ factorial (n $-$ 1)
has already been discussed in detail in Section 1.6.3.2.

Definition 2.2.2.3:

Exponentiation Function Exp: is a function of two variables x and n where x is any non-negative real number and n is an integer **(*though n can be taken as non-integer also, but we restrict to integers only*)**

Exp (x, n) denoted by x^n, is defined recursively as follows:

For n = 0

$$\text{Exp}(x, 0) = x^0 = 1$$

For n > 0

$$\text{Exp}(x, n) = x \times \text{Exp}(x, n-1)$$

i.e

$$x^n = x \times x^{n-1}$$

For n < 0, let n = $-$ m for m > 0

$$x^n = x^{-m} = \frac{1}{x^m}$$

In x^n, n is also called the exponent/power of x.

For example: if x = 1.5, n = 3, then

Also, Exp (1.5, 3) = $(1.5)^3 = (1.5) \times [(1.5)^2] = (1.5)\,[1.5 \times (1.5)^1]$
$= 1.5\,[\,(1.5 \times (1.5 \times (1.5)^0))]$
$= 1.5[(1.5 \times (1.5 \times 1))] = 1.5\,[(1.5 \times 1.5)]$
$= 1.5\,[2.25] = 3.375$

$$\text{Exp}(1.5, -3) = (1.5)^{-3} = \frac{1}{(1.5)^3} = \frac{1}{3.375}$$

Further, the following rules apply to exponential function.

For two integers m and n and a real number b the following identities hold:

$$\left((b)^m\right)^n = b^{mn}$$
$$\left(b^m\right)^n = \left(b^n\right)^m$$
$$b^m \,.\, b^n = b^{m+n}$$

For b ≥ 1 and for all n, the function b^n is monotonically increasing in n. In other words, if $n_1 \geq n_2$ then $b^{n_1} \geq b^{n_2}$ if b ≥ 1.

Definition 2.2.2.4:

Polynomial: A polynomial in n of degree k, where k is a non-negative integer, over R, the set of real numbers, denoted by P(n), is of the form

$$P_k(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0,$$

where $a_k \neq 0$ and $a_i \in R$, i = 0, 1, …, k.

Using the summation notation

$$P_k(n) = \sum_{i+0}^{k} a_i\, n^i \qquad a_k \neq 0, \quad a_i \in R$$

Each of $(a_i\, n^i)$ **is called a term.**

Generally the suffix k** in **$P_k(n)$ is dropped and in stead of $P_k(n)$ we write P(n) only

We may note that $P(n) = n^k = 1.n^k$ for any k, is a single-term polynomial. If $k \geq 0$ then $P(n) = n^k$ is monotonically increasing. Further, if $k \leq 0$ then $p(n) = n^k$ is monotonically decreasing.

Notation: Though 0^o is not defined, yet, unless otherwise mentioned, we will take $0^o = 1$. The following is a very useful result relating the exponentials and polynomials

Result 2.2.2.5: For any constants b and c with b > 1

$$\lim_{n\to\infty} \frac{n^c}{b^n} = 0$$

The result, in non-mathematical terms, states that for any given constants b and c, but with b > 1, the terms in the sequence $\frac{1^c}{b^1}, \frac{2^c}{b^2}, \frac{3^c}{b^3}, \ldots, \frac{k^c}{b^k}, \ldots.$ gradually decrease and approaches zero. **Which further means that for constants b and c, and integer variable n, the exponential term b^n, for b > 1, increases at a much faster rate than the polynomial term n^c.**

Definition 2.2.2.6:

The **letter e** is used to denote the quantity

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots\ldots,$$

and is taken as the base of natural logarithm function, then for all real numbers x,

we define the exponential function

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots. = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

For, all real numbers, we have $e^x \geq 1 + x$

Further, if $|x| \leq 1$ then $1+x \leq e^x \leq 1 + x + x^2$

The following is another useful result, which we state without proof:

Result 2.2.2.7: $\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x$

Definition 2.2.2.8:

Logarithm: The concept of logarithm is defined indirectly through the definition of Exponential defined earlier. If a > 0, b > 0 and c > 0 are three real numbers, such that

$$c = a^b$$

Then $b = \log_a c$ (*read as log of c to the base a*)

Then a is called the base of the algorithm.

For example: if $2^6 = 64$, the $\log_2 64 = 6$,

i.e., 2 raised to power 6 gives 64.

Generally two bases viz 2 and e are very common in scientific and computing fields and hence, the following **specials notations** for these bases are used:

(i)	lg n	denotes $\log_2 n$	(base 2)
(ii)	ln n	denotes $\log_e n$	(base e);

where the *letter l* in *ln* denotes *logarithm* and the *letter n* in *ln* denotes *natural.*

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

Result 2.2.2.9:

For n, a natural number and real numbers a, b and c all greater than 0, the following identities are true:

(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) $\log_a b = \dfrac{1}{\log_b a}$

(vi) $a^{\log_b c} = c^{\log_b a}$

2.3 MATHEMATICAL EXPECTATION

In average-case analysis of algorithms, to be discussed in Unit 3, 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% ($\because$ *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

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

2.4 PRINCIPLE OF MATHEMATICAL INDUCTION

Frequently, in establishing the truth of a set of statements, where each of the statement is a function of a natural number n, we will be using the Principle of Mathematical Induction. In view of the significance of the principle, we discuss it here briefly.

The principle is quite useful in establishing the **correctness of a countably many infinite number of statements**, through **only finite number of steps.**

The method consists of the following three major steps:

1. **Verifying/establishing** the correctness of the **Base** Case
2. **Assumption** of Induction Hypothesis for the given set of statements
3. **Verifying/Establishing** the Induction Step

We explain the involved ideas through the following example:

[*] 10 is the average of the class boundaries 0 and 20.

Let us consider the following sequence in which nth term S(n) is the sum of first (n−1) powers of 2, e.g.,

$S(1) = 2^0 = 2 - 1$
$S(2) = 2^0 + 2^1 = 2^2 - 1$
$S(3) = 2^0 + 2^1 + 2^2 = 2^3 - 1$

We (*intuitively*) feel that

S(n) = $2^0 + 2^1 + \ldots + 2^{n-1}$ should be $2^n - 1$ for all $n \geq 1$.
We may establish the correctness of the intuition, i.e., correctness of all the **infinite** number of statements

$S(n) = 2^n - 1$ for all $n \geq 1$,

through only the following **three** steps:

(i) **Base Case:** (*In this example, n = 1*) we need to show that
$S(1) = 2^1 - 1 = 1$

But, by definition $S(1) = 2^0 = 1 = 2 - 1 = 2^1 - 1$ is correct

(ii) **Induction Hypothesis:** **Assume**, for some k > base-value (*=1, in this case*) that
$S(k) = 2^k - 1$.

(iii) **Induction Step:** Using (i) & (ii) establish that (*in this case*)

$S(k+1) = 2^{k+1} - 1$

In order to establish
$S(k+1) = 2^{k+1} - 1,$ (A)
we use the definition of S(n) and Steps (i) and (ii) above

By definition

$S(k+1) = 2^0 + 2^1 + \ldots + 2^{k+1-1}$
$= (2^0 + 2^1 + \ldots + 2^{k-1}) + 2^k$ (B)

But by definition
$2^0 + 2^1 + \ldots + 2^{k-1} = S(k).$ (C)

Using (C) in (B) we get
$S(k+1) = S(k) + 2^k$ (D)

and by Step (ii) above
$S(k) = 2^k - 1$ (E)

Using (E) in (D), we get

$S(k+1) = (2^k - 1) + 2^k$
$\therefore S(k+1) = 2.2^k - 1 = 2^{k+1} - 1$
which establishes (A).

Ex.1) By using Principle of Mathematical Induction, show that 6 divides $n^3 - n$, where n is a non-negative integer.

Ex.2) Let us assume that we have unlimited supply of postage stamps of Rs. 5 and Rs. 6 then

(i) through, direct calculations, find what amounts can be realized in terms of only these stamps.

(ii) Prove, using Principle of Mathematical Induction, the result of your efforts in part (i) above.

2.5 CONCEPT OF EFFICIENCY OF AN ALGORITHM

If a problem is algorithmically solvable then it may have more than one algorithmic solutions. In order to choose the best out of the available solutions, there are criteria for making such a choice. The complexity/efficiency measures or criteria are based on requirement of computer resources by each of the available solutions. The solution which takes *least resources* is taken as the *best solution* and is generally chosen for solving the problem. However, it is very difficult to even enumerate all possible computer resources e.g., time taken by the *designer* of the solution and the time taken by the *programmer* to encode the algorithm.

Mainly the two computer resources taken into consideration for efficiency measures, are *time and space* requirements for *executing* the program corresponding to the solution/algorithm. *Until it is mentioned otherwise, we will restrict to only time complexities of algorithms of the problems.*

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.

It is easy to realize that given an algorithm for multiplying two n $\times$ n matrices, the time required by the algorithm for finding the product of two 2 $\times$ 2 matrices, is expected to take much less time than the time taken by the same algorithm for multiplying say two 100 $\times$ 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.

Formally, one of the definitions of the ***size of an instance*** *of a problem may be taken as the* ***number of bits*** *required in representing the 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 problem. For example,

(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.

There are *two approaches* for determining complexity (or time required) for executing an algorithm, viz.,

(i) empirical (*or a posteriori*) and
(ii) theoretical (*or a priori*).

In the **empirical approach** (the programmed) algorithm is *actually* executed on various *instances* of the *problem* and the *size* (s) and *time* (*t*) of execution for each instance is noted. And then by some numerical or other technique, *t* is determined as a function of s. This function then, is taken as complexity of the *algorithm* under consideration.

In the theoretical approach, we *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* and each of the basic operations like '+', comparison of pair of integers etc., is assumed to take one or more, but some constant number of, (basic) units of time for execution. Time for execution for each *structuring rule*, is assumed to be some function of the times required for the constituents of the structure.
Thus starting from the basic instructions and operations and using structuring rules, one can calculate the time complexity of a program or an algorithm.

The theoretical approach has a number of advantages over the empirical approach including the ones enumerated below:

(i) The approach does not depend on the programming language in which the algorithm is coded and on how it is coded in the language,

(ii) The approach does not depend on the computer system used for executing (a programmed version of) the algorithm.

(iii) In case of a comparatively inefficient algorithm, which ultimately is to be rejected, the computer resources and programming efforts which otherwise would have been required and wasted, will be saved.

(iv) 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 under consideration. In the case of theoretical approach, the size n may *be arbitrarily large*. However, in empirical approach, because of practical considerations, only the instances of *moderate sizes* may be considered.

Remark 2.5.1:

In view of the advantages of the theoretical approach, we are going to use it as *the only approach* for computing complexities of algorithms. As mentioned earlier, in the approach, no particular computer is taken into consideration for calculating time complexity. But different computers have different execution speeds. However, the speed of one computer is generally some constant multiple of the speed of the other.

Therefore, this fact of differences in the speeds of computers by constant multiples is taken care of, in the complexity functions *t* for general instance sizes *n*, by writing the complexity function as *c.t(n)* where c is an arbitrary constant.

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.* Thus, functions differing from each other by constant factors, when treated as time complexities should not be treated as different, i.e., should be treated as complexity-wise equivalent.

Remark 2.5.2:

Asymptotic Considerations:

Computers are generally used to solve problems involving *complex* solutions. The complexity of solutions may be either because of the large number of involved computational steps and/or because of large size of input data. The plausibility of the claim apparently follows from the fact that, when required, computers are used *generally not to* find the product of two 2×2 matrices *but to find* the product of two $n \times n$ matrices for large n, running into hundreds or even thousands.

Similarly, computers, when required, are generally used *not only to find roots* of quadratic equations but for finding roots of complex equations including polynomial equations of *degrees more than hundreds or sometimes even thousands.*

The above discussion leads to the conclusion that when considering time complexities $f_1(n)$ and $f_2(n)$ of (computer) solutions of a problem of size n, we need to consider and compare the behaviours of the two functions only for large values of n. If the relative behaviours of two functions for smaller values conflict with the relative behaviours for larger values, then we may ignore the conflicting behaviour for smaller values. For example, if the earlier considered two functions

$$f_1(n) = 1000\, n^2 \quad \text{and}$$
$$f_2(n) = 5n^4$$

represent time complexities of two solutions of a problem of size n, then despite the fact that

$$f_1\,(n) \geq f_2\,(n) \qquad \text{for } n \leq 14,$$

we would still prefer the solution having f_1 (n) as time complexity because

$$f_1(n) \leq f_2\,(n) \qquad \text{for all } n \geq 15.$$

This explains the reason for the presence of the phrase ‘$n \geq k$’ in the definitions of the various measures of complexities and approximation functions, discussed below:

Remark 2.5.3:

Comparative Efficiencies of Algorithms: Linear, Quadratic, Polynomial Exponential

Suppose, for a given problem P, we have two algorithms say A_1 and A_2 which solve the given problem P. Further, assume that we also know time-complexities $T_1(n)$ and T_2 (n) of the two algorithms for problem size n. How do we know which of the two algorithms A_1 and A_2 is better?

The difficulty in answering the question arises from the difficulty in comparing time complexities $T_1(n)$ and $T_2(n)$.

For example, let $\mathbf{T_1(n) = 1000n^2}$ and $\mathbf{T_2(n) = 5n^4}$
Then, for all n, neither $T_1(n) \leq T_2(n)$ nor $T_2(n) \leq T_1$ (n).

More explicitly

$$T_1\,(n) \geq T_2\,(n) \text{ for } n \leq 14 \quad \text{and}$$
$$T_1\,(n) \leq T_2\,(n) \text{ for } n \geq 15.$$

The issue will be discussed in more detail in Unit 3. However, here we may mention that, in view of the fact that we generally use computers to solve problems of *large* *sizes,* **in the above case, the algorithms A_1 with time-complexity $T_1(n) = 1000n^2$ is preferred over the algorithm A_2 with time-complexity $T_2(n) = 5n^4$, because $T_1(n) \leq T_2(n)$ for all $n \geq 15$.**

In general if a problem is solved by two algorithms say B_1 and B_2 with time-complexities $BT_1(N)$ and $BT_2(n)$ respectively, then

(i) **if** $BT_1(n)$ is a *polynomial* in n, i.e,

$$BT_1(n) = a_k n^k + a_{k-1} + \ldots + a_i n^i + \ldots + a_1 n + a_0$$

for some $k \geq 0$ with a_i's as real numbers and $a_k > 0$, and

$BT_2(n)$ is an *exponential* function of n, i.e., $BT_2(n)$ is of the form
$BT_2(n) = c\, a^n$ where c and a are some real numbers with $a > 1$,
then generally, for large values of n, $BT_1(n) \leq BT_2(n)$.
Hence, algorithm B_1 with *polynomial time* complexity is assumed to the more efficient and is preferred over algorithm B_2 with *exponential time* complexity.

(ii) If, again a problem is solved by two algorithms D_1 and D_2 with respectively polynomial time complexities DT_1 and DT_2 then if

$$\text{degree}(DT_1) < \text{degree}(DT_2),$$

then the algorithm D_1 is assumed to be more efficient and is preferred over D_2.

Certain complexity functions occur so frequently that special names commensurate with their usage may be given to such functions. For example, complexity function *c n* is called *linear time complexity* and corresponding algorithm, is called as *linear algorithm*.

Similarly, the terms '***quadratic***' *and* '**polynomial time**' *complexity functions and algorithms* are used when the involved complexity functions are respectively of the forms $c\, n^2$ and $c_1 n^k + \ldots\ldots + c_k$.

In the next section we find examples of linear and quadratic algorithms.

Remark 2.5.4:

For all practical purposes, *the use of c, in (c t(n)) as time complexity measure*, offsets properly the effect of differences in the speeds of computers. *However, we need to be on the guard, because in some rarely occurring situations, neglecting the effect of c may be misleading.*

For example, if two algorithms A_1 and A_2 respectively take n^2 *days* and n^3 *secs* for execution of an instance of size n of a particular problem. *But a 'day' is a constant multiple of a 'second'.* Therefore, as per our conventions we may take the two complexities as of $C_2 n^2$ *and* $C_3 n^3$ for some constants C_2 and C_3. As, we will discuss later, the algorithm A_1 taking $C_2 n^2$ time is *theoretically* preferred over the algorithm A_2 with time complexity $C_3 n^3$. The preference is based on asymptotic behaviour of complexity functions of the algorithms. *However* in this case, *only* for instances requiring millions of years, the algorithm A_1 requiring $C_2 n^2$ time outperforms algorithms A_2 requiring $C_3 n^3$.

Remark 2.2.5:

Unit of Size for Space Complexity: Though most of the literature discusses the complexity of an algorithm only in terms of expected time of execution, generally neglecting the space complexity. However, space complexity has one big advantage over time complexity.

*In the case of space complexity, unit of measurement of space requirement can be well defined as a **bit.** But in the case of time complexity such obvious choice is not available. The problem lies in the fact that unlike 'bit' for space, there is no standard time unit such as 'second' since, we do not have any standard computer to which single time measuring unit is applicable.*

Ex.3) For a given problem P, two algorithms A_1 and A_2 have respectively time complexities T_1 (n) and T_2 (n) in terms of size n, where

T_1 (n) = $4n^5 + 3n$ and
T_2 (n) = $2500n^3 + 4n$

Find the range for n, the size of an instance of the given problem, for whichA_1 is more efficient than A_2.

2.6 WELL KNOWN ASYMPTOTIC FUNCTIONS & NOTATIONS

We often want to know a quantity only approximately, and not necessarily exactly, just to compare with another quantity. And, in many situations, correct comparison may be possible even with approximate values of the quantities. The advantage of the possibility of correct comparisons through even approximate values of quantities, is that the time required to find approximate values may be much less than the times required to find exact values.

We will introduce five approximation functions and their notations.

The purpose of these asymptotic growth rate functions to be introduced, is to facilitate the recognition of *essential character of a complexity function* through some simpler functions delivered by these notations. For example, a complexity function f(n) = $5004\ n^3 + 83\ n^2 + 19\ n + 408$, has essentially the same behaviour as that of $g(n) = n^3$ as the problem size n becomes larger and larger. But $g(n) = n^3$ is much more comprehensible and its value easier to compute than the function f(n).

2.6.1 Enumerate the five well-known approximation functions and how these are pronounced

(i) O: (O (n^2) is pronounced as 'big-oh of n^2' or sometimes just as oh of n^2)

(ii) Ω: (Ω (n^2) is pronounced as 'big-omega of n^2 or sometimes just as omega of n^2')

(iii) Θ: (Θ (n^2) is pronounced as 'theta of n^2')

(iv) o: (o (n^2) is pronounced as 'little-oh of n^2')

(v) ω: (ω (n^2) is pronounced as 'little-omega of n^2').

The O-notation was introduced by Paul Bachman in his book Analytische Zahlentheorie (1894)

These approximations denote relations from functions to functions.

For example, if functions

$f, g: N \rightarrow N$ are given by

$f(n) = n^2 - 5n$ and
$g(n) = n^2$

then

$O(f(n)) = g(n)$ or $O(n^2 - 5n) = n^2$

To be more precise, each of these notations is a mapping that associates a *set of* functions to each function under consideration. For example, if f (n) is a polynomial of degree k then the set O (f (n)) includes all polynomials of degree less than or equal to k.

Remark 2.6.1.1:

In the discussion of any one of the five notations, generally two functions say f and g are involved. The functions have their domains and codomains as N, the set of natural numbers, i.e.,

$f: N \rightarrow N$
$g: N \rightarrow N$

These functions may also be considered as having domain and codomain as R.

2.6.2 The Notation O

Provides asymptotic *upper bound* for a given function. Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number constants C and k such that

$$f(x) \leq C\, g(x) \quad \text{for all } x \geq k \qquad \text{(A)}$$

(*The restriction of being positive on integers/reals is justified as all complexities are positive numbers*).

Example 2.6.2.1: For the function defined by

$f(x) = 2x^3 + 3x^2 + 1$
show that

(i) $f(x) = O(x^3)$
(ii) $f(x) = O(x^4)$
(iii) $x^3 = O(f(x))$
(iv) $x^4 \neq O(f(x))$
(v) $f(x) \neq O(x^2)$

Solution:

Part (i)

Consider

$$f(x) = 2x^3 + 3x^2 + 1$$
$$\leq 2x^3 + 3x^3 + 1\ x^3 = 6x^3 \qquad \text{for all } x \geq 1$$

(by replacing each term x^i by the highest degree term x^3)

$\therefore$ there exist C = 6 and k = 1 such that

$f(x) \le C.\ x^3$ for all $x \ge k$

Thus we have found the required constants C and k. Hence f(x) is $O(x^3)$.

Part (ii)

As above, we can show that

$f(x) \le 6\ \ x^4$ for all $x \ge 1$.

However, we may also, by computing some values of f(x) and x^4, find C and k as follows:

$f(1) = 2+3+1 = 6$; $(1)^4 = 1$
$f(2) = 2.2^3 + 3.2^2 + 1 = 29$; $(2)^4 = 16$
$f(3) = 2.3^3 + 3.3^2 + 1 = 82$; $(3)^4 = 81$

for C = 2 and k = 3 we have
$f(x) \le 2.\ x^4$ for all $x \ge k$

Hence f(x) is $O(x^4)$.

Part (iii)

for C = 1 and k = 1 we get
$x^3 \le C\ (2x^3 + 3x^2 +1)$ for all $x \ge k$

Part (iv)

We prove the result by contradiction. Let there exist positive constants C and k such that

$x^4 \le C\ (2x^3 + 3x^2 +1)$ for all $x \ge k$
$\therefore x^4 \le C\ (2x^3 +3x^3+x^3) = 6Cx^3$ for $x \ge k$
$\therefore\ x^4 \le 6\ C\ x^3$ for all $x \ge k$.

implying $x \le 6C$ for all $x \ge k$
But for x = max of { 6 C + 1, k}, the previous statement is not true.
Hence the proof.

Part (v)

Again we establish the result by contradiction.

Let $O\ (2\ x^3+3x^2+1) = x^2$

Then for some positive numbers C and k

$2x^3 + 3x^2+1 \le C\ x^2$ for all $x \ge k$,

implying

$x^3 \le C\ x^2$ for all $x \ge k$ *(Θ $x^3 \le 2x^3+3x^2+1$ for all $x \ge 1$)*

implying

$x \le C$ for $x \ge k$

Again for x = max $\{C+1, k\}$

The last inequality does not hold. Hence the result.

Example 2.6.2.2:

The big-oh notation can be used to estimate S_n, the sum of first n positive integers

Hint: $S_n=1+2+3+\ldots\ldots\ldots+n \le n+n+\ldots\ldots\ldots\ldots+n = n^2$

Therefore, $S_n = O(n^2)$.

Remark 2.6.2.2:

It can be easily seen that for given functions f(x) and g(x), if there exists one pair of C and k with $f(x) \le C.g(x)$ for all $x \ge k$, then there exist infinitely many pairs (C_i, k_i) which satisfy

$$f(x) \le C_i\, g(x) \qquad \text{for all } x \ge k_i.$$

Because for **any C_i** $\ge$ C and **any k_i** $\ge$ k, the above inequality is true, if $f(x) \le c.g(x)$ for all $x \ge k$.

2.6.3 The Ω Notation

The Ω Notation provides an asymptotic *lower bound* for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be Ω (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

$$f(x) \ge C\,(g(x)) \qquad \text{whenever } x \ge k$$

Example 2.6.3.1:

For the functions

$$f(x) = 2x^3 + 3x^2 + 1 \text{ and } h(x) = 2x^3 - 3x^2 + 2$$

show that

(i) $f(x) = \Omega(x^3)$

(ii) $h(x) = \Omega(x^3)$

(iii) $h(x) = \Omega(x^2)$

(iv) $x^3 = \Omega(h(x))$

(v) $x^2 \ne \Omega(h(x))$

Solutions:

Part (i)

For C =1, we have
$f(x) \ge C\, x^3$ for all $x \ge 1$

Part (ii)

$h(x) = 2x^3 - 3x^2 + 2$
Let C and $k > 0$ be such that
$2x^3 - 3x^2 + 2 \geq C\,x^3$ for all $x \geq k$
i.e., $(2 - C)\,x^3 - 3x^2 + 2 \geq 0$ for all $x \geq k$

Then $C = 1$ and $k \geq 3$ satisfy the last inequality.

Part (iii)

$$2x^3 - 3x^2 + 2 = \Omega\,(x^2)$$

Let the above equation be true.

Then there exists positive numbers C and k
s.t.

$2x^3 - 3x^2 + 2 \geq C\,x^2$ for all $x \geq k$
$2\,x^3 - (3 + C)\,x^2 + 2 \geq 0$

It can be easily seen that lesser the value of C, better the chances of the above inequality being true. So, to begin with, let us take $C = 1$ and try to find a value of k s.t

$2x^3 - 4x^2 + 2 \geq 0.$
For $x \geq 2$, the above inequality holds

$\therefore$ k=2 is such that

$2x^3 - 4x^2 + 2 \geq 0$ for all $x \geq k$

Part (iv)

Let the equality

$$x^3 = \Omega\,(2x^3 - 3x^2 + 2)$$

be true. Therefore, let $C > 0$ and $k > 0$ be such that

$$x^3 \geq C(2(x^3 - 3/2\,x^2 + 1))$$

For $C = ½$ and $k = 1$, the above inequality is true.

Part (v)

We prove the result by contradiction.

Let $x^2 = \Omega\,(3x^3 - 2x^2 + 2)$

Then, there exist positive constants C and k such that

$x^2 \geq C\,(3x^3 - 2x^2 + 2)$ for all $x \geq k$

i.e., $(2C + 1)\,x^2 \geq 3C\,x^3 + 2 \geq C\,x^3$ for all $x \geq k$

$\frac{2C+1}{C} \geq x$ for all $x \geq k$

But for any $x \geq 2\dfrac{(2C+1)}{C}$,

The above inequality can not hold. Hence contradiction.

2.6.4 The Notation Θ

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be Θ (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that
$C_2\, g(x) \leq f(x) \leq C_1\, g(x)$ for all $x \geq k$.

(*Note the last inequalities represent two conditions to be satisfied simultaneously viz., $C_2\, g(x) \leq f(x)$ and $f(x) \leq C_1\, g(x)$*)

We state the following theorem without proof, which relates the three functions O, Ω, Θ.

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)).

Examples 2.6.4.1: For the function
$f(x) = 2x^3 + 3x^2 + 1$,
show that

(i) $f(x) = \Theta(x^3)$

(ii) $f(x) \neq \Theta(x^2)$

(iii) $f(x) \neq \Theta(x^4)$

Solutions

Part (i)

for $C_1 = 3$, $C_2 = 1$ and $k = 4$

1. $C_2\, x^3 \leq f(x) \leq C_1\, x^3$ for all $x \geq k$

Part (ii)

We can show by contradiction that no C_1 exists.

Let, if possible for some positive integers k and C_1, we have $2x^3+3x^2+1 \leq C_1 .\, x^2$ for all $x \geq k$

Then

$x^3 \leq C_1\, x^2$ for all $x \geq k$

i.e.,

$x \leq C_1$ for all $x \geq k$

But for

$x = \max \{C_1 + 1, k\}$

The last inequality is not true.

Part (iii)

$f(x) \neq \Theta(x^4)$

We can show by contradiction that there does not exist C_2 s.t

$C_2 x^4 \leq (2x^3 + 3x^2 + 1)$

If such a C_2 exists for some k then $C_2 x^4 \leq 2x^3 + 3x^2 + 1 \leq 6x^3$ for all $x \geq k \geq 1$,

implying

$C_2 x \leq 6$ for all $x \geq k$

But for $x = \left(\frac{6}{C_2} + 1 \right)$

the above inequality is false. Hence, proof of the claim by contradiction.

2.6.5 The Notation o

The asymptotic upper bound provided by big-oh notation may or may not be tight in the sense that if $f(x) = 2x^3 + 3x^2 + 1$

Then for $f(x) = O(x^3)$, though there exist C and k such that

$f(x) \leq C(x^3)$ for all $x \geq k$

yet there may also be some values for which the following equality also holds

$f(x) = C(x^3)$ for $x \geq k$

However, if we consider

$f(x) = O(x^4)$

then there can not exist positive integer C s.t

$f(x) = C x^4$ for all $x \geq k$

The case of $f(x) = O(x^4)$, provides an example for the next notation of small-oh.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers.

Further, let $C > 0$ **be any number**, then $f(x) = o(g(x))$ (pronounced as little oh of g of x) if there exists natural number k satisfying

$f(x) < C\, g(x)$ for all $x \geq k \geq 1$ (B)

Here we may note the following points

(i) In the case of little-oh the constant C does not depend on the two functions f (x) and g (x). Rather, we can *arbitrarily* choose C >0

(ii) The inequality (B) is strict whereas the inequality (A) of big-oh is not necessarily strict.

Example 2.6.5.1: For $f(x) = 2x^3 + 3x^2 + 1$, we have

(i) $f(x) = o(x^n)$ for any $n \geq 4$.

(ii) $f(x) \neq o(x^n)$ for $n \leq 3$

Solutions:

Part (i)

Let C > 0 **be given and** to find out k satisfying the requirement of little-oh.
Consider

$$2x^3 + 3x^2 + 1 < C\ x^n$$
$$= 2 + \frac{3}{x} + \frac{1}{x^3} < C\ x^{n-3}$$

Case when n = 4

Then above inequality becomes

$$2 + \frac{3}{x} + \frac{1}{x^3} < C\ x$$

if we take $k = \max\left\{\frac{7}{C}, 1\right\}$

then

$$2x^3 + 3x^2 + 1 < C\,x^4 \qquad \text{for } x \geq k.$$

In general, as $x^n > x^4$ for $n \geq 4$,

therefore

$2x^3 + 3x^2 + 1 < C\ x^n$ for $n \geq 4$
for all $x \geq k$
with $k = \max\left\{\frac{7}{c}, 1\right\}$

Part (ii)

We prove the result by contradiction. Let, if possible, $f(x) = 0(x^n)$ for $n \leq 3$.

Then there exist positive constants C and k such that $2x^3+3x^2+1< C\ x^n$ for all $x \geq k$.

Dividing by x^3 throughout, we get

$$2 + \frac{3}{x} + \frac{1}{x^2} < C\,x^{n-3}$$

$n \leq 3$ and $x \geq k$

As C is arbitrary, we take

C = 1, then the above inequality reduces to

$$2+\frac{3}{x}+\frac{1}{x^2} < C.\ x^{n-3} \quad \text{for } n \leq 3 \text{ and } x \geq k \geq 1.$$

Also, it can be easily seen that

$$x^{n-3} \leq 1 \qquad \text{for } n \leq 3 \text{ and } x \geq k \geq 1.$$

$$\therefore\ 2+\frac{3}{x}+\frac{1}{x^2} \leq 1 \qquad \text{for } n \leq 3$$

However, the last inequality is not true. Therefore, the proof by contradiction.

Generalising the above example, we get the

Example 2.6.5.3: If f(x) is a polynomial of degree m and g(x) is a polynomial of degree n. Then

$f(x) = o(g(x))$ if and only if $n>m$.

We state (without proof) below two results which can be useful in finding small-oh upper bound for a given function.

More generally, we have

Theorem 2.6.5.3: Let f(x) and g(x) be functions in definition of small-oh notation.

Then f(x) = o(g(x) if and only if

$$\lim_{x\to\infty}\frac{f(x)}{g(x)} = 0$$

Next, we introduce the last asymptotic notation, namely, small-omega. The relation of small-omega to big-omega is similar to what is the relation of small-oh to big-oh.

2.6.6 The Notation ω

Again the asymptotic lower bound Ω may or may not be tight. However, the asymptotic bound ω *cannot* be tight. *The formal definition of ω is follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.

Further

Let C > 0 **be any number**, then

$f(x) = \omega\ (g(x))$

if there exist a positive integer k s.t

$f(x) > C\ \ g(x)$ for all $x \geq k$

Example 2.6.6.1:

If $f(x) = 2x^3 + 3x^2 + 1$

then

$f(x) = \omega(x)$

and also

$f(x) = \omega(x^2)$

Solution:

Let C be any positive constant.

Consider

$2x^3 + 3x^2 + 1 > C\,x$

To find out $k \geq 1$ satisfying the conditions of the bound ω.

$2x^2 + 3x + \frac{1}{x} > C$ *(dividing throughout by x)*

Let k be integer with $k \geq C+1$

Then for all $x \geq k$

$2x^2 + 3x + \frac{1}{x} \geq 2x^2 + 3x > 2k^2 + 3k > 2C^2 + 3C > C.$ (Θ $k \geq C+1$)

$\therefore f(x) = \omega(x)$

Again, consider, for any $C > 0$,

$2x^3 + 3x^2 + 1 > C\,x^2$

then

$2x + 3 + \frac{1}{x^2} > C$ Let k be integer with $k \geq C+1$

Then for $x \geq k$ we have

$2x + 3 + \frac{1}{x^2} \geq 2x + 3 > 2k + 3 > 2C + 3 > C$

Hence

$f(x) = \omega(x^2)$

In general, we have the following two theorems (stated without proof).

Theorem 2.6.6.2: If f(x) is a polynomial of degree n, and g(x) is a polynomial of degree n, then

$f(x) = \omega(g(x))$ if and only if $m > n$.

More generally

Theorem 2.6.6.3: Let f(x) and g(x) be functions in the definitions of little-omega Then $f(x) = \omega(g(x))$ if and only if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \infty$$

$$\lim_{x \to \infty} \frac{g(x)}{f(x)} = 0$$

Ex.4) Show that $n! = O(n^n)$.

Ex.5) Show that $n^2 + 3\log n = O(n^2)$.

Ex.6) Show that $2^n = O(5^n)$.

2.7 SUMMARY

In this unit, first of all, a number of mathematical concepts are defined. We defined the concepts of function, 1-1 function, onto function, ceiling and floor functions, mod function, exponentiation function and log function. Also, we introduced some mathematical notations.

In Section 2.3, the concept of mathematical expectation is introduced, which is useful in average-case analysis. Formally, mathematical expectation is defined as follows:

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

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

Also, the Principle of Mathematical Induction is discussed, which is useful in establishing truth of infinitely many statements.
The method of Mathematical Induction consists of the following three major steps:

4. **Verifying/establishing** the correctness of the **Base** Case
5. **Assumption** of Induction Hypothesis for the given set of statements
6. **Verifying/Establishing** the Induction Step

Complexity of an algorithm: There are two approaches to discussing and computing complexity of an algorithm viz

(i) empirical

(ii) theoretical.

These approaches are briefly discussed in Section 2.5. However, in the rest of the course *only theoretical* approach is used for the analysis and computations of complexity of algorithms. Also, it is mentioned that analysis and computation of complexity of an algorithm may be considered in terms of

(i) *time* expected to be taken or

(ii) *space* expected to be required for executing the algorithm.

Again, we will throughout consider *only the time complexity*. Next, the concepts of linear, quadratic, polynomial and exponential time complexities and algorithms, are discussed.

Next, five **Well Known Asymptotic Growth Rate functions are defined and corresponding notations are introduced. Some important results involving these are stated and/or proved**

The notation O provides asymptotic *upper bound* for a given function.
Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.
Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number Constants C and k such that

$f(x) \leq C\, g(x)$ for all $x \geq k$

The Ω notation provides an asymptolic *lower bound* for a given function

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be Ω (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

$f(x) \geq C\,(g(x))$ whenever $x \geq k$

The Notation Θ

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be Θ (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that
$C_2\, g(x) \leq f(x) \leq C_1\, g(x)$ for all $x \geq k$.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers.

Further, let C > 0 **be any number**, then f(x) = o(g(x)) (pronounced as little oh of g of x) if there exists natural number k satisfying

$f(x) < C\, g(x)$ for all $x \geq k \geq 1$

The Notation ω

Again the asymptotic lower bound Ω may or may not be tight. However, the asymptotic bound ω *cannot* be tight. *The formal definition of ω is as follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.

Further

Let C > 0 **be any number**, then

$f(x) = \omega\,(g(x))$

if there exist a positive integer k s.t

$f(x) > C\;\; g(x)$ for all $x \geq k$

2.8 SOLUTIONS/ANSWERS

Ex. 1) We follow the three-step method explained earlier.
Let S(n) be the statement: 6 divides $n^3 - n$

Base Case: For n = 0, we **establish** S(0); i.e., we establish that 6 divides $0^3 - 0 = 0$

But $0 = 6 \times 0$. Therefore, 6 divides 0. Hence S(0) is correct.

Induction Hypothesis: For any positive integer k **assume** that S(k) is correct, i.e., assume that 6 divides $(k^3 - k)$.

Induction Step: Using the conclusions/assumptions of earlier steps, to **show** the correctness of S(k+1), i.e., to *show* that 6 divides $(k+1)^3 - (k+1)$.

Consider $(k+1)^3 - (k+1) = (k^3 + 3k^2 + 3k + 1) - (k + 1)$
$= k^3 + 3k^2 + 2k = k(k+1)(k+2)$

If we show that k (k+1) (k+2) is divisible by 6 than by Principle of Mathematical Induction the result follows.

Next, we prove that 6 divides k (k+1) (k+2) for all non-negative integers.

As k, (k+1) and (k+2) are three consecutive integers, therefore, at least one of these is even, i.e., divisible by 2. Hence it remains to show that k(k+1) (k+2) is divisible by 3. This we establish through the following **case analysis:**

(i) If k is divisible by 3, then, of course, k(k+1) (k+2) is also divisible by 3.

(ii) If on division of k, remainder is 1, i.e., if k = 3t + 1 for some integer t then k (k+1) (k+2) = (3t+1) (3t+2) (3t+3) = 3 (3t+1) (3t+2) (t + 1) is divisible by 3.

(iii) If on division of k, remainder is 2, i.e., if k = 3t + 2 then k(k+1) (k+2) = (3t + 2) (3t + 3) (3t + 4) = 3(3t + 2) (t +1) (3t + 4) is again divisible by 3.

Ex. 2) **Part (i):** With stamps of Rs. 5 and Rs. 6, we can make the following the following amounts

5	=	$1 \times 5 + 0 \times 6$	
6	=	$0 \times 5 + 1 \times 6$	using 2 stamps

10	=	$2 \times 5 + 0 \times 6$	
11	=	$1 \times 5 + 1 \times 6$	using 2 stamps
12	=	$0 \times 5 + 2 \times 6$	

15	=	$3 \times 5 + 0 \times 6$	
16	=	$2 \times 5 + 1 \times 6$	
17	=	$1 \times 5 + 2 \times 6$	using 3 stamps
18	=	$0 \times 5 + 3 \times 6$	

19 is not possible

20	=	$4 \times 5 + 0 \times 6$	
21	=	$3 \times 5 + 1 \times 6$	
22	=	$2 \times 5 + 2 \times 6$	using 4 stamps
23	=	$1 \times 5 + 3 \times 6$	
24	=	$0 \times 5 + 4 \times 6$	

25	=	$5\times 5 + 0\times 6$	
26	=	$4\times 5 + 1\times 6$	
27	=	$3\times 5 + 2\times 6$	
28	=	$2\times 5 + 3\times 6$	using 5 stamps
29	=	$1\times 5 + 4\times 6$	
30	=	$0\times 5 + 5\times 6$	

It appears that for any amount $A \geq 20$, it can be realized through stamps of only Rs. 5 and Rs. 6.

Part (ii): We attempt to show using Principle of Mathematical Induction, that any amount of Rs. 20 or more can be realized through using a number of stamps of Rs. 5 and Rs. 6.

Base Step: For **amount = 20,** we have shown earlier that $20 = 4.5 + 0.6$.

Hence, Rs. 20 can be realized in terms of stamps of Rs. 5 and Rs. 6.

Induction Hypothesis: Let for some $\mathbf{k \geq 20}$, amount k can be realized in terms of some stamps of Rs. 5 and some stamps of Rs. 6.

Induction Step: Next, the case of the amount k+1 is **analysed** as follows:

Case (i): If at least one Rs. 5 stamp is used in making an amount of Rs. k, then we replace one Rs. 5 stamp by one Rs. 6 stamp to get an amount of Rs. (k+1). Hence the result in this case.

Case (ii): If all the 'Stamps' in realizing Rs. $k \geq 20$ is through only Rs. 6 stamps, then k must be at least 24 and hence at least 4 stamps must be used in realzing k. Replace 4 of Rs. 6 stamps by 5 of Rs. 5 stamps. So that out of amount k, $6 \times 4 = 24$ are reduced through removing of 4 stamps of Rs. 6 and an amount of Rs. $5 \times 5 = 25$ is added. Thus we get an amount of $k - 24 + 25 = k+1$. This completes the proof.

Ex. 3) Algorithm A_1 is more efficient than A_2 for those values of n for which

$$4n^5 + 3n = T_1(n) \leq T_2(n) = 2500\, n^3 + 4n$$

i.e.,

$$4n^4 + 3 \leq 2500\, n^2 + 4$$

i.e.,

$$4n^2 - 2500 \leq 1/n^2 \qquad \text{(i)}$$

Consider

$$4n^2 - 2500 = 0$$

then $n = 50/2 = 25$

for $n \leq 25$

$$4n^2 - 2500 \leq (25)^2 - 2500 \leq 0 \leq 1/n^2$$

Hence (i) is satisfy

Next, consider $n \geq 26$

$$4n^2 - 2500 \geq 4(26)^2 - 2500 = 2704 - 2500$$
$$= 204 > 1 > \frac{1}{(26)^2} \geq \frac{1}{n^2}$$

Therefore, **for n ≥ 26, (i)is not satisfied**

Conclusion: For problem sizes n, with $1 \leq n \leq 25$, A_1 is more efficient than A_2. However, for $n \geq 25$, A_2 is more efficient than A_1

Ex. 4)

$n!/n^n = (n/n)\ ((n-1)/n)\ ((n-2)/n)\ ((n-3)/n)\ldots(2/n)(1/n)$
$= 1(1-(1/n))\ (1-(2/n))\ (1-(3/n))\ldots(2/n)(1/n)$

Each factor on the right hand side is less than equal to 1 for all value of n. Hence, The right hand side expression is always less than one.

Therefore, $n!/n^n \leq 1$
or, $n! \leq n^n$
Therefore, $n! = O(n^n)$

Ex. 5)

For large value of n, $3\log n << n^2$

Therefore, $3\log n/ n^2 << 1$

$(n^2 + 3\log n)/ n^2 = 1 + 3\log n/ n^2$

or, $(n^2 + 3\log n)/ n^2 < 2$

or, $n^2 + 3\log n = O(n^2)$.

Ex.6) We have, $2^n/5^n < 1$
or, $2^n < 5^n$
Therefore, $2^n = O(5^n)$.

2.9 FURTHER READINGS

1. *Discrete Mathematics and Its Applications* (*Fifth Edition*) K.N. Rosen: Tata McGraw-Hill (2003).

2. *Introduction to Alogrithms* (*Second Edition*), T.H. Coremen, C.E. Leiserson & C. Stein: Prentice – Hall of India (2002).