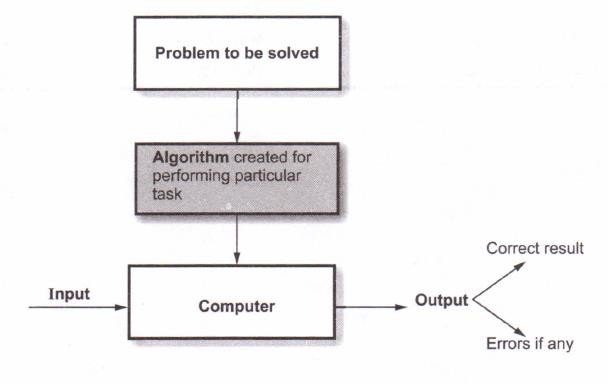
**UNIT -1**

**Introduction: What is an Algorithm, Algorithm Specification, Pseudocode Conventions, Recursive Algorithm, Performance Analysis, Space Complexity, Time Complexity, Amortized Complexity, Asymptotic Notation, Practical Complexities, and Performance Measurement.**

**Algorithm :**

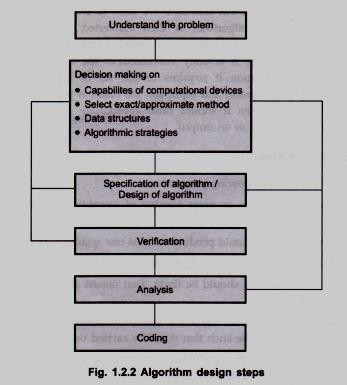
The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

This notion of algorithm is represented in the following Fig.



After understanding the problem statement we have to create an algorithm carefully for the given problem. The algorithm is then converted into some programming language and then given to some computing device (computer). The computer then executes this algorithm which is actually submitted in the form of source program. During the process of execution it requires certain set of input. With the help of algorithm (in the form of program) and input set, the result is produced as an output. If the given input is invalid then it should raise appropriate error message; otherwise correct result will be produced as an output.

**Properties of Algorithm :** Algorithm should posses following properties

1. **Input -** The input of zero or more number of quantities should be given to the algorithm.
2. **Output -** The algorithm should produce at least one quantity, as an output.
3. **Definiteness -** Each instruction in algorithm should be specific and unambiguous.
4. **Finiteness -** The algorithm should be finite. That means after finite number of steps it should terminate.
5. **Effectiveness -** Every step of algorithm should be feasible. In other words, every step of algorithm should be such that it can be carried out by pen and pencil.

The implementation of such algorithms can be done by programming language. Hence the term program can be defined as –

# "Program is the expression of algorithm using some programming language". Design of Algorithm

Various steps for algorithm design are- These steps are -

1. Understanding the problem
2. Decision making on
   1. Capabilities of computational devices
   2. Choice for either exact or approximate problem solving method
   3. Data structures
   4. Algorithmic strategies.
3. Specification of algorithm
4. Algorithmic verification
5. Analysis of algorithm
6. Implementation or coding of algorithm.

Let us now discuss each step in detail

# Understanding the problem

This is the very first step in designing of algorithm. In this step first of all you need to **understand the problem** statement completely. While understanding the problem statements, read the problem description carefully and ask questions for clarifying the doubts about the problem. But there are some types of problems that are commonly occurring and to solve such problems there are typical algorithms which are already available. Normally you have to design **an algorithm on** your own.

After carefully understanding the problem statements find out what are the necessary inputs for solving that problem. The input to the algorithm is called instance of the problem. It is very important to decide the range of inputs so that the boundary values of algorithm get fixed. The algorithm should work correctly for all valid inputs.

# Decision making

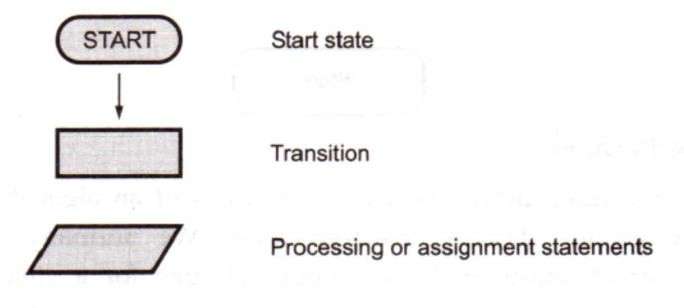
After finding the required input set for the given problem we have to analyze the input and need to decide certain issues such as capabilities of computational devices, whether to use exact or approximate problem solving, which data structures has to be used, and to find the algorithmic technique for solving the given problem. This step serves as a base for the actual design of algorithm.

* 1. Capabilities of computational devices
  2. Choice for either exact or approximate problem solving method
  3. Data structures
  4. Algorithmic strategies

“Algorithmic strategies” is a general approach by which many problems can be solved algorithmically. These problems may belong to different areas of computing. Algorithmic strategies are also called as algorithmic techniques or algorithmic paradigm.

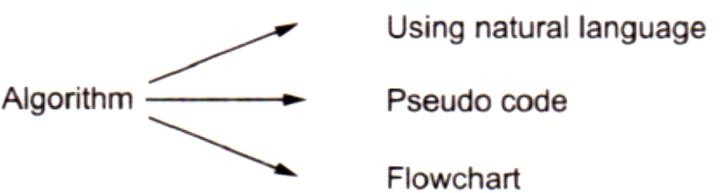
Algorithm Design Techniques -

* *Brute force:* This is straight forward technique with naive approach.
* *Divide-and-conquer:* The problem is divided into smaller instances.
* *Dynamic programming:* The results of smaller, re-occuring instances are obtained to solve the problem.
* *Greedy technique:* To solve the problem locally optimal decisions are made.
* *Back tracking:* This method is based on the trial and error. If we want to solve some problem then desired solution is chosen from the finite set S.



# Specification of algorithm

There are various ways by which we can specify an algorithm.



It is very simple to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear, and thereby we get brief specification.

## For example: *Write an algorithm to perform addition of two numbers. (using Natural* Language)

Step 1: Read the first number say a. Step 2: Read the second number say b.

Step 3: Add the two numbers and store the result in a variable c. Step 4: Display the result.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of pseudo code. Pseudo code is nothing but a combination of natural language and programming language constructs. A pseudo code is usually more precise than a natural language.

## For example: *Write an algorithm for performing addition of two numbers. (Using Pseudo* code)

Algorithm sum ( a , b )

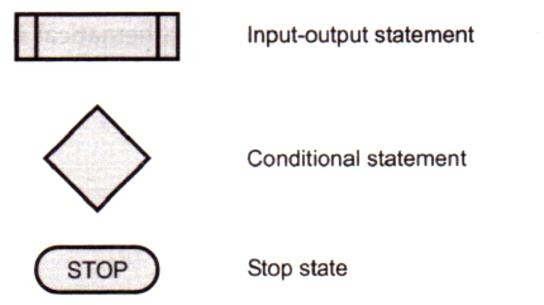
//Problem Description This algorithm performs addition of

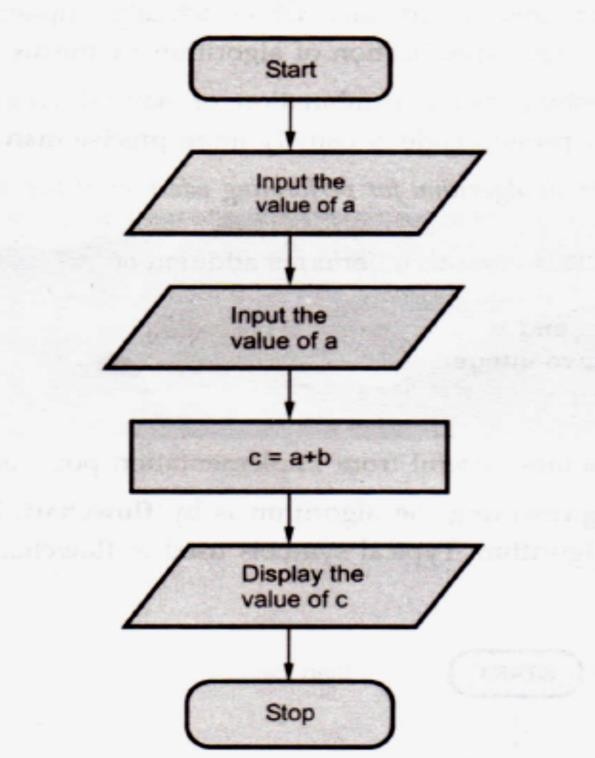
//two integers

//Input: two integers a and b

//Output : addition of two integers c ← a+b

write ( c )

This specification is more useful from implementation point of view. Another way of representing the algorithm is by flowchart. Flowchart is a graphical representation of an algorithm. Typical symbols used in flowchart are –

For Example :

# Algorithmic verification

Algorithmic verification means checking correctness of an algorithm. After specifying an algorithm we go for checking its correctness. We normally check whether the algorithm gives correct output in finite amount of time for a valid set of input. The proof of correctness of an algorithm can be complex sometimes. A common method of proving the correctness of an algorithm is by using mathematical induction.

# Analysis of algorithm

While analyzing an algorithm we should consider following factors -

* Time efficiency of an algorithm
* Space efficiency of an algorithm
* Simplicity of an algorithm
* Generality of an algorithm
* Range of input.

# Implementation of algorithm

The implementation of an algorithm is done by suitable programming language. For example, if an algorithm consists of objects and related methods then it will be better to implement such algorithm using some object oriented programming language like C++ .or

JAVA. While writing a program for given algorithm it is essential to write an optimized code. This will reduce the burden on compiler.

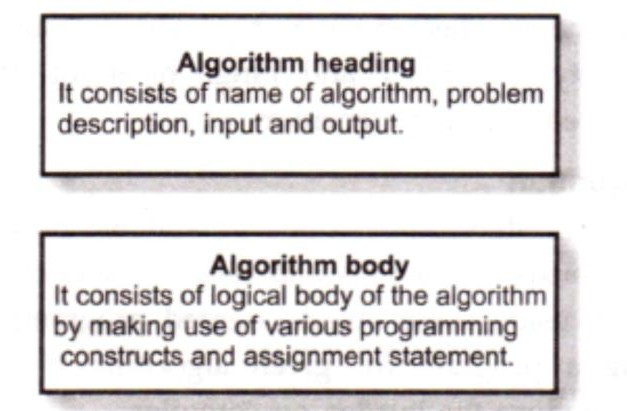
# Testing a Program

* Testing a program is an activity carried out to expose as many errors as possible and to correct them.
* There are two phases for testing a program :

1. Debugging
2. Profiling

# Pseudo Code for Expressing Algorithm

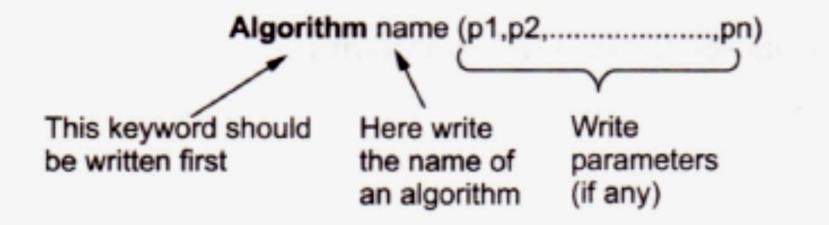
Algorithm is basically a sequence of instructions written in simple English language. The algorithm is broadly divided into two sections –



Let us understand some rules for writing the algorithm.

# Pseudo-Code Conventions:

1. Algorithm is a procedure consisting of heading and body. The heading consists of keyword Algorithm and name of the algorithm and parameter list. The syntax is



1. Then in the heading section we should write following things :

// Problem Description :

// Input:

// Output:

1. Then body of an algorithm is written, in which various programming constructs like if, for, while or some assignment statements may be written.
2. The compound statements should be enclosed within { and }brackets.
3. Single line comments are written using // as beginning of comment.
4. The identifier should begin by letter and not by digit. An identifier can be a combination of alphanumeric string.

It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself. Basic data types used are integer, float, char, Boolean and so on. The pointer type is also used to point memory location. The compound data type such as structure or record can also be used.

1. Using assignment operator f- an assignment statement can be given. For instance : Variable ← expression
2. There are other types of operators such as Boolean operators such as true or false. Logical operators such as **and, or, not**. And relational operators such as *<,* <=, >, >=, =, ≠
3. The array indices are stored with in square brackets '[' ‘]’. The index of array usually start at zero. The multidimensional arrays can also be used in algorithm.
4. The inputting and outputting can be done using read and write. For example :

write("This message will be displayed on console") ;

**read(**val);

1. The conditional statements such as if-then or if-then-else are written in following form:

**if** (condition) **then** statement

**if** (condition) **then** statement **else** statement

If the **if-then** statement is of compound type then { and } should be used for enclosing block.

1. while statement can be written as :

**while** (condition) **do**

{

statement 1

statement 2

:::::

:::::

}

While the condition is true the block enclosed with { statement after } will be executed.

1. The general form for writing for loop is : for variable ← value1 to valuen do

{

statement 1

statement 2

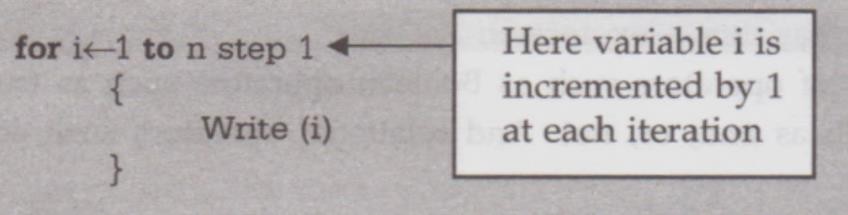
:

statement n

}

Here value1 is initialization condition and valuen is a terminating condition.

Sometime a keyword step is used to denote increment or decrement the value of variable for example



1. The repeat - until statement can be written as : repeat

statement 1

statement 2 statement n

until (condition)

1. The break statement is used to exit from inner loop. The return statement is used to return control from one point to another. Generally used while exiting from function.

Note that statements in an algorithm executes in sequential order i.e. in the same order as they appear-one after the other.

**Some Examples**

**Example 1 :** Write an algorithm to count the sum of n numbers.

# Algorithm sum (1, n)

//problem Description : This algorithm is for finding the

//sum of given n numbers

*//*Input : 1 to n numbers

//Output : The sum of n numbers result ← 0

**for** i ← 1 to n **do** i ← i + 1 result ← result + i

**return** result

**Example** 2 : Write an algorithm to check whether given number is even or odd.

**Algorithm** eventest (val)

//problem Description : This algorithm test whether given

//number is even or odd

//Input : the number to be tested i.e. val

//Output : Appropriate messages indicating even or oddness if (val%2=0) **then**

write ("Given number is even")

else

write ("Given number is odd")

**Example** 3 : Write an algorithm for sorting the elements.

**Algorithm** sort ( a, n)

//problem Description : sorting the elements in ascending order

//Input : An array a in which the elements are stored and n

*//is* total number of elements in the array

//Output : The sorted array

**for** i ← 1to n **do**

**for** j ← i + 1**to** n – 1 **do**

{

**if** (a([i]>a[j] ) **then**

{

temp ← a[i] a[i] ← a[j] a[j] ← temp

}

}

**write** ("List is sorted")

**Example** 4 : Write an algorithm to find factorial of n number.

**Algorithm** fact (n)

//Problem Description : This algorithm finds the factorial *of* given number n

//Input : The number n of which the factorial is to be calculated.

//Output : factorial value of given n number.

**if** (n ← 1) **then**

return 1

else

return n \* fact(n - 1)

**Example 5 :** Write an algorithm to perform multiplication of two matrices.

**Algorithm** Mul(A,B,n)

//Problem Description : This algorithm is for computing multiplication of two matrices

//Input : The two matrices A,B and order of them as n

//Output : The multiplication result will be in matrix C for i ←1 to n do

for j ← 1 to n do

C[i,j] ← 0

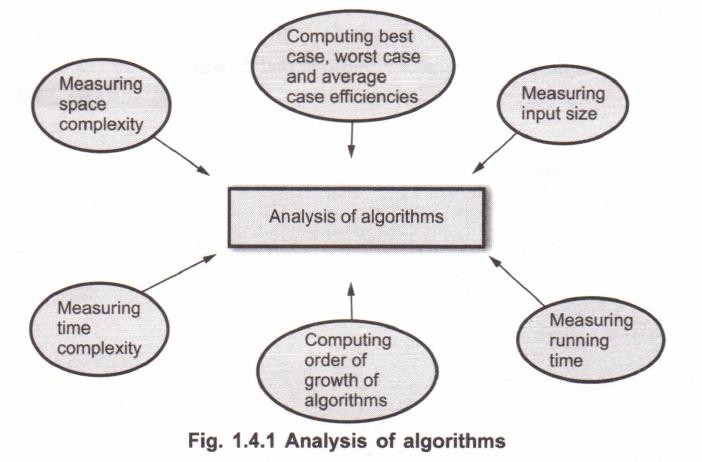
for k ← 1 to n do

C[ i , j ] ← C[ i , j ] +A[ i , k ] \* B[ k , j ]

**Performance Analysis**

The efficiency of an algorithm can be decided by measuring the performance of an algorithm. We can measure the performance of an algorithm by computing two factors.

* 1. Amount of time required by an algorithm to execute.
  2. Amount of storage required by an algorithm.

This is popularly known as time complexity and space complexity of an algorithm.

**Space Complexity**

The space complexity can be defined as amount of memory required by an algorithm

to run.

To compute the space complexity we use two factors : constant and instance characteristics. The space requirement S(p) can be given as :

S(p) = C + Sp

where C is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers. And Sp is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance.

# Consider three examples of algorithms to compute the space complexity.

Example 1:

# Algorithm Add (a, b, c)

//**Problem Description:** This algorithm computes the addition of three elements

**//Input :** a.b, and c are of floating type

**//Output :** The addition is returned

**return** a + b + c

The space requirement for algorithm given in Example1is S(p) = C Ѳ (Sp) = 0

If we assume that a, b and c occupy one word size then total size comes to be 3.

# Example 2 :

**Algorithm Add (x, n)**

**//Problem Description :** The algorithm performs addition of

**//all the elements in an array**. Array is of floating type.

**//Input :** An array x and n is total number of elements in array

**//Output :** returns sum which is of data type float.

Sum ← 0.0

**for** ← i ← 1 to n **do**

sum ← sum + x[i]

**return** sum

The space requirement for the above given algorithm is - S(p) >= (n + 3)

The 'n' space required for *x[ ],* one unit space for *n,* one unit for *i* and one unit for *sum.*

**Example** 3 :

# Algorithm: ArraySum (x, n)

**//Problem Description :** This is a recursive algorithm which

//computes addition of all the elements in an array x[]

**//Input:** x[i] is of floating type, total number of elements in an array

**//Output :** returns addition of n elements of an array

**return** (ArraySum(x, n-l ) + x[n-1])

The space requirement is - S(p) ← 3(n + 1)

The internal stack used for recursion includes space for formal parameters, local variables and return address. The space required by each call to function Add requires at least three words (space for n values + space for return address + pointer to x [ ]. The depth of recursion in n+1 (n times call to function and one return call). The recursion stack space will be >= 3(n+1).

# Time Complexity

The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.

# It is difficult to compute the time complexity in terms of physically clocked time.

For instance in multiuser system, executing time depends on many factors such as -

* System load
* Number of other programs running
* Instruction set used
* Speed of underlying hardware.

The time complexity is therefore given in terms of **frequency count.**

**Frequency count** is a count denoting number of times of execution of statement.

# Measuring an Input Size

It is observed that if the input size is longer, then usually algorithm runs for a longer time. Hence we can compute the efficiency of an algorithm as a function to which input size is passed as a parameter. Sometimes to implement an algorithm we require prior knowledge of input size. For example, while performing multiplication of two matrices we should know order of these matrices. Then only we can enter the elements of matrices. Sometimes the input size is taken as an approximate value. For example, in spell checking algorithms we can predict exact size of the input.

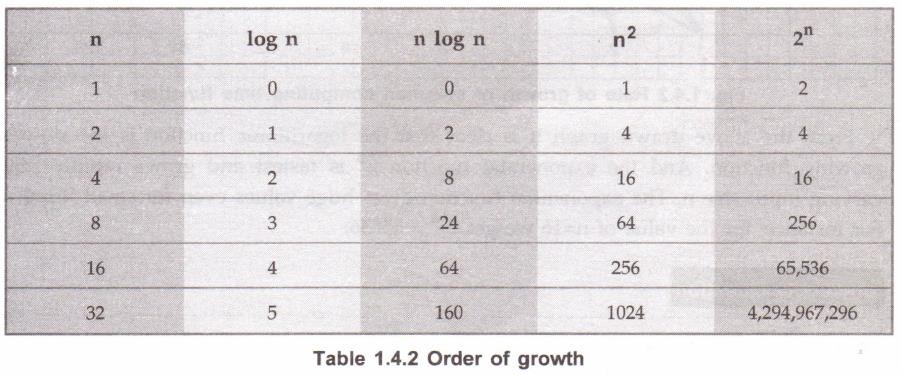
# Measuring Running Time

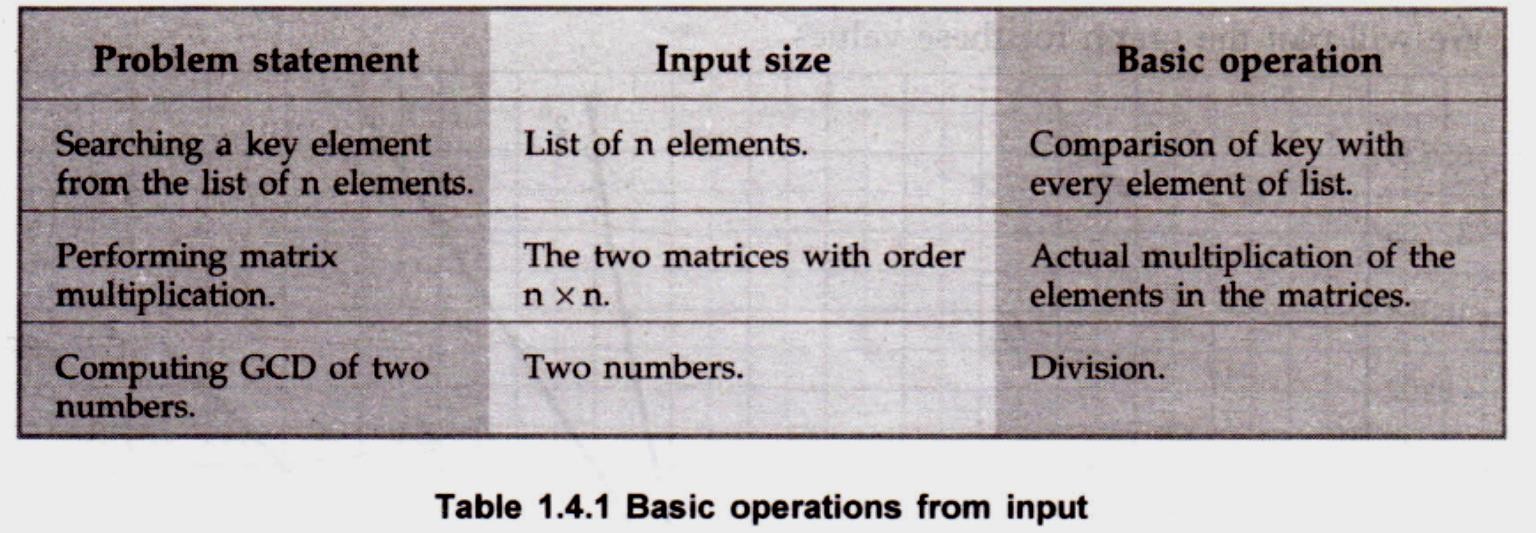
We have already discussed that the time complexity is measured in terms of a unit called frequency count. The time which is measured for analyzing an algorithm is generally running time.

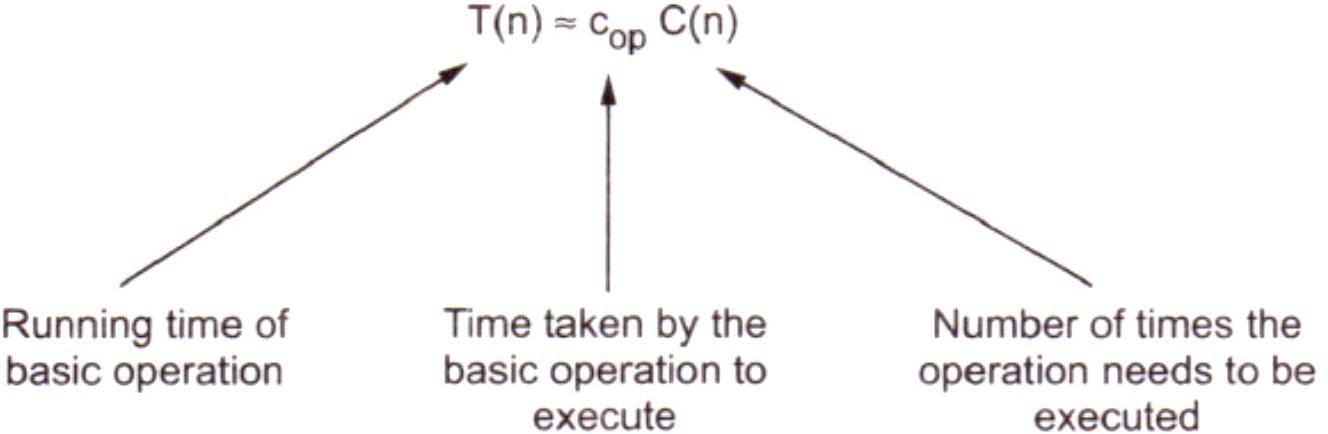
# From an algorithm :

* We first identify the important operation (core logic) of an algorithm. This operation is called the basic operation.
* It is not difficult to identify basic operation from an algorithm. Generally the operation which is more time consuming is a basic operation in the algorithm. Normally such basic operation is located in inner loop. For example in sorting algorithm the operation which is comparing the elements and then placing them at appropriate locations is a basic operation. The concept of basic operations can be well understood with the help of following example.

13

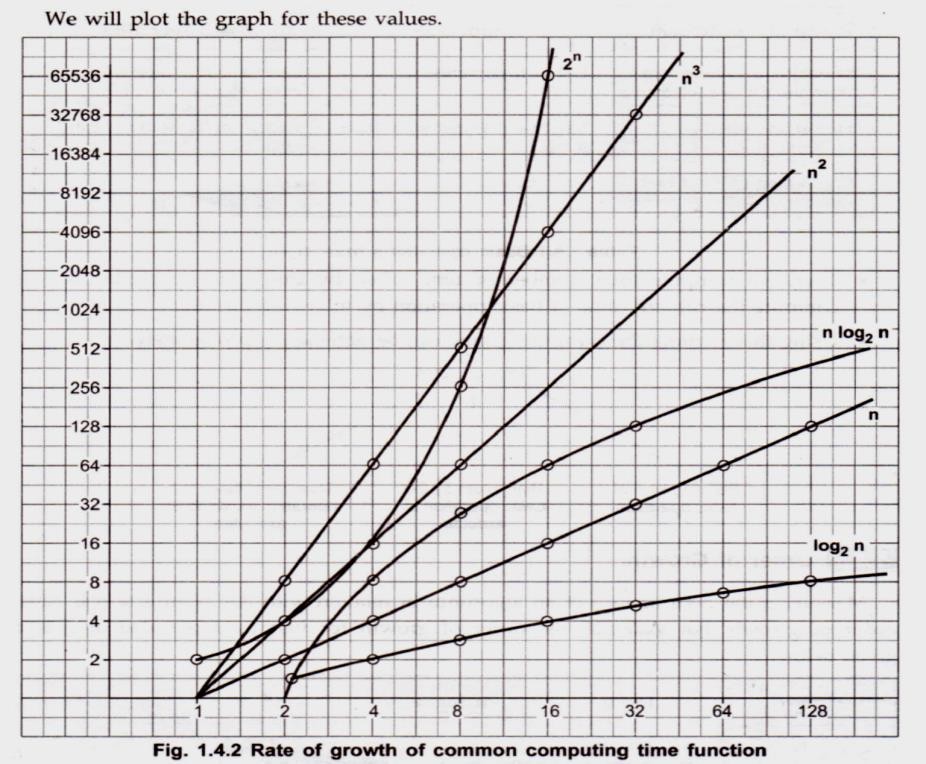




* Then we compute total number of time taken by this basic operation. We can compute the running time of basic operation by following formula.
* Using this formula the computing time can be obtained approximately.

# Order of Growth

Measuring the performance of an algorithm in relation with the input size n is called **order of growth.** For example, the order of growth for varying input size of n is as given below.



From the above drawn graph it is clear that the logarithmic function is the slowest growing function. And the exponential function 2n is fastest and grows rapidly with varying input size n. The exponential function gives huge values even for small input n. For instance: for the value of n=16 we get 216 = 65536.

# Asymptotic Notations

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.

Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time". Various notations such as Ω, Ѳ and O used are called asymptotic notions.

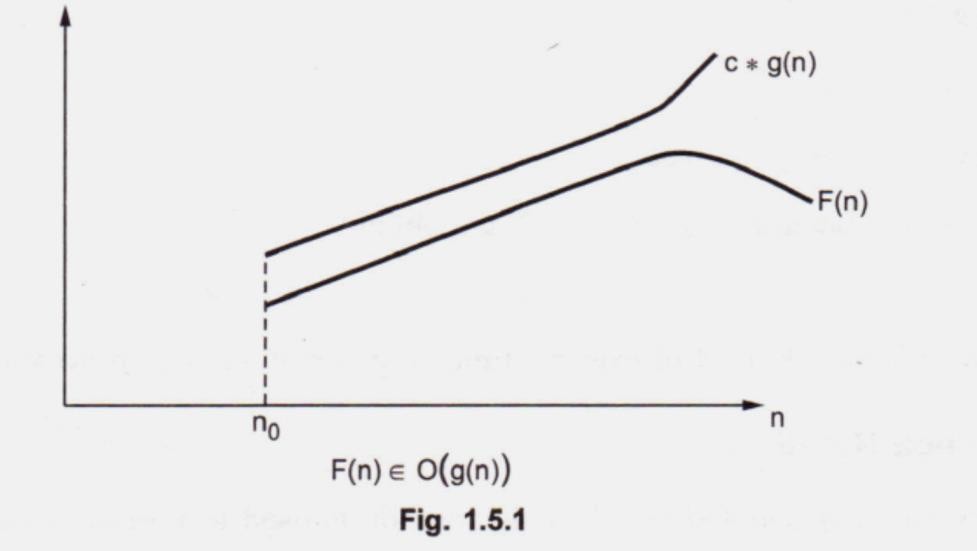
# Big oh Notation (O)

The Big oh notation is denoted by 'O'. It is a method of representing the upper bound of algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

# Definition

Let F(n) and g(n) be two non-negative functions.

Let n0 and constant c are two integers such that no denotes some value of input and n>n0. Similarly c is some constant such that c > 0. We can write

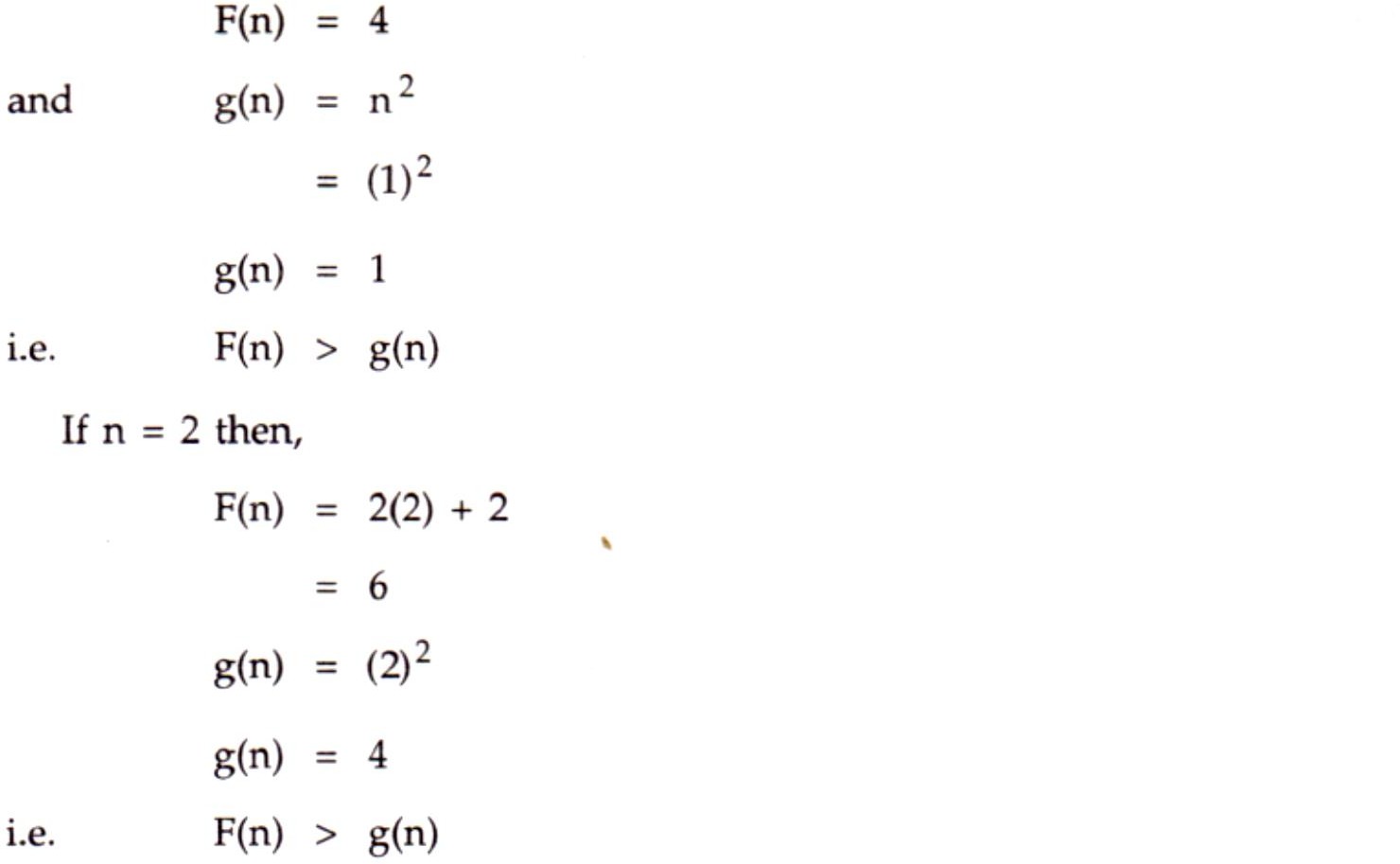


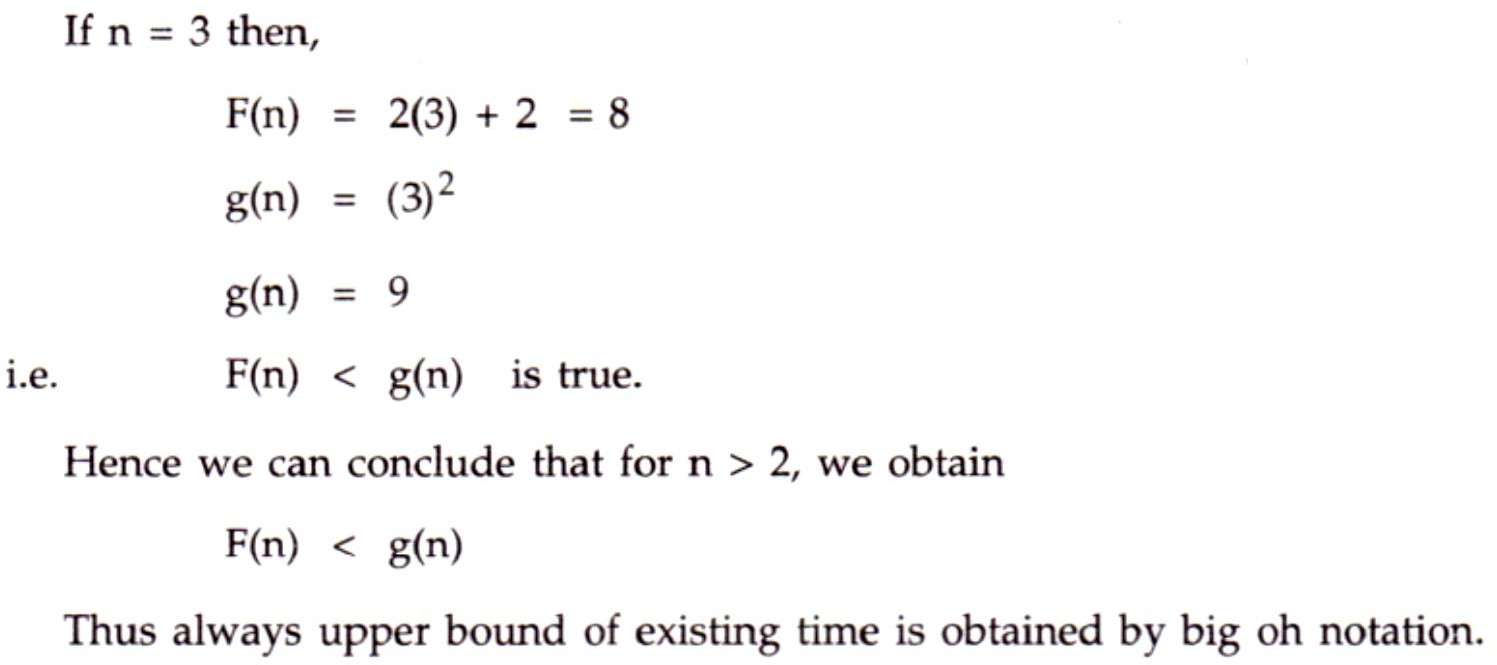
F(n) <= c\*g(n)

then F(n) is big oh of g(n). It is also denoted as F(n) E a (g(n)). In other words F(n) is less than g(n) if g(n) is multiple of some constant c.

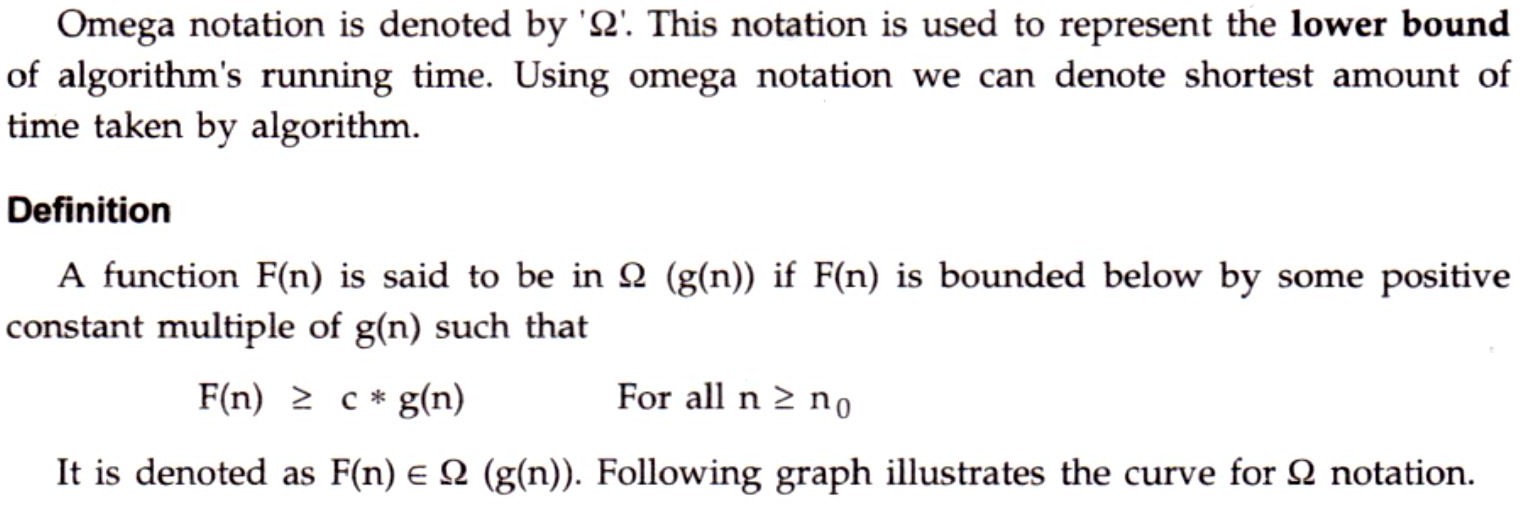
**Example :** Consider function F(n) = 2n + 2 and g(n) = n2. Then we have to find some constant c, so that F(n) <= c \* g(n). As F(n) = 2n + 2 and g(n) = n2 then we find c for n = 1 then

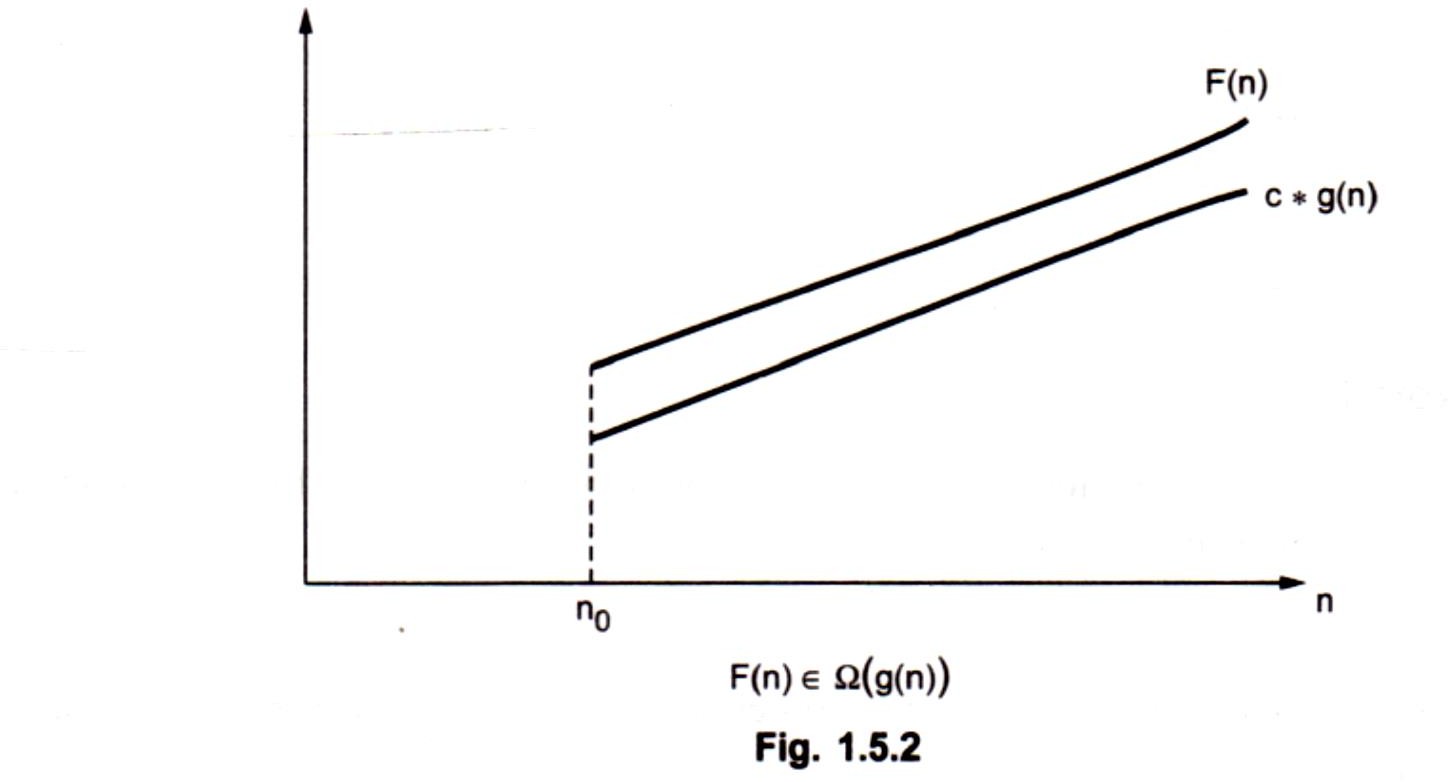
F(n) = 2n + 2 = 2(1) + 2

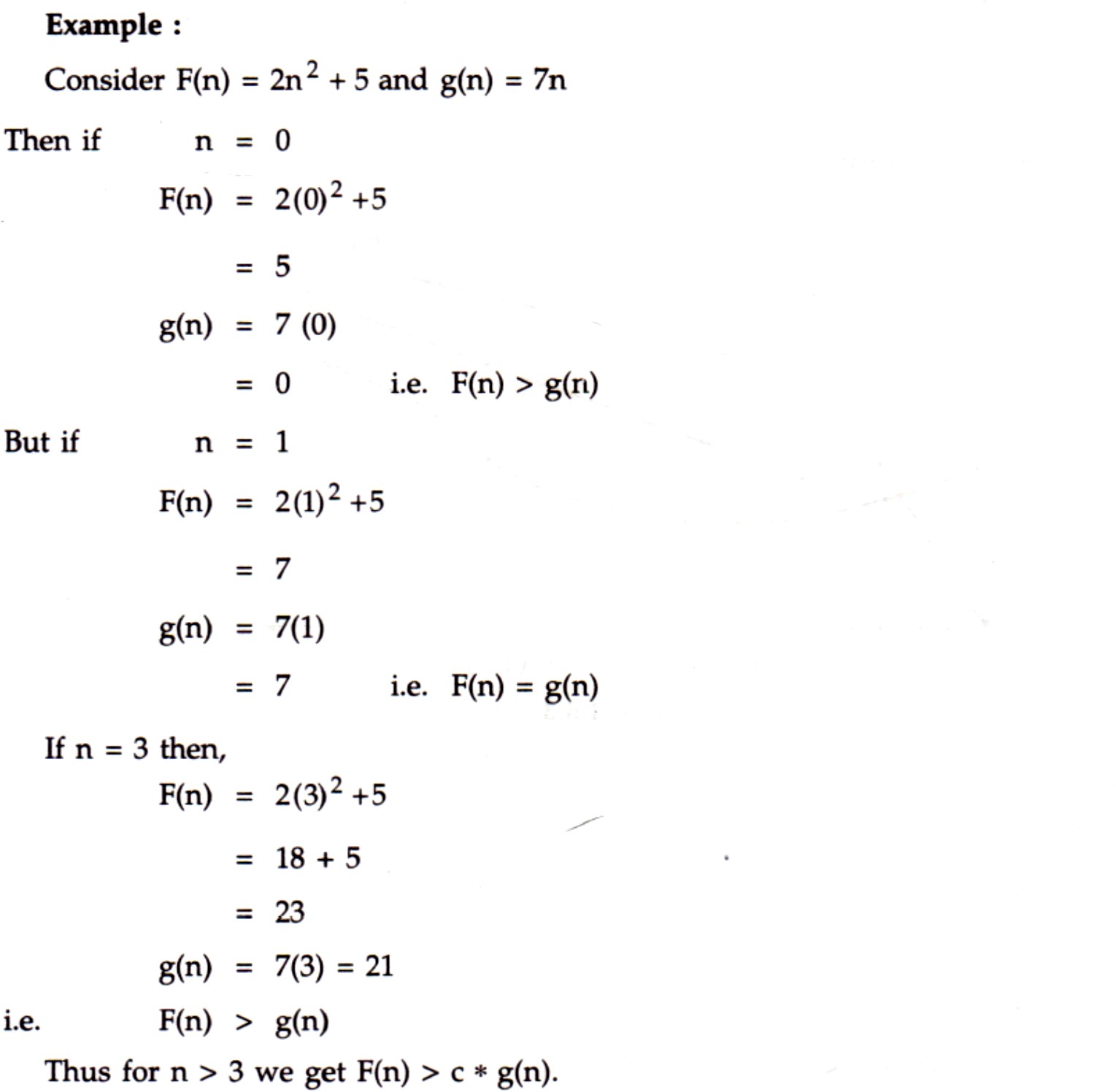


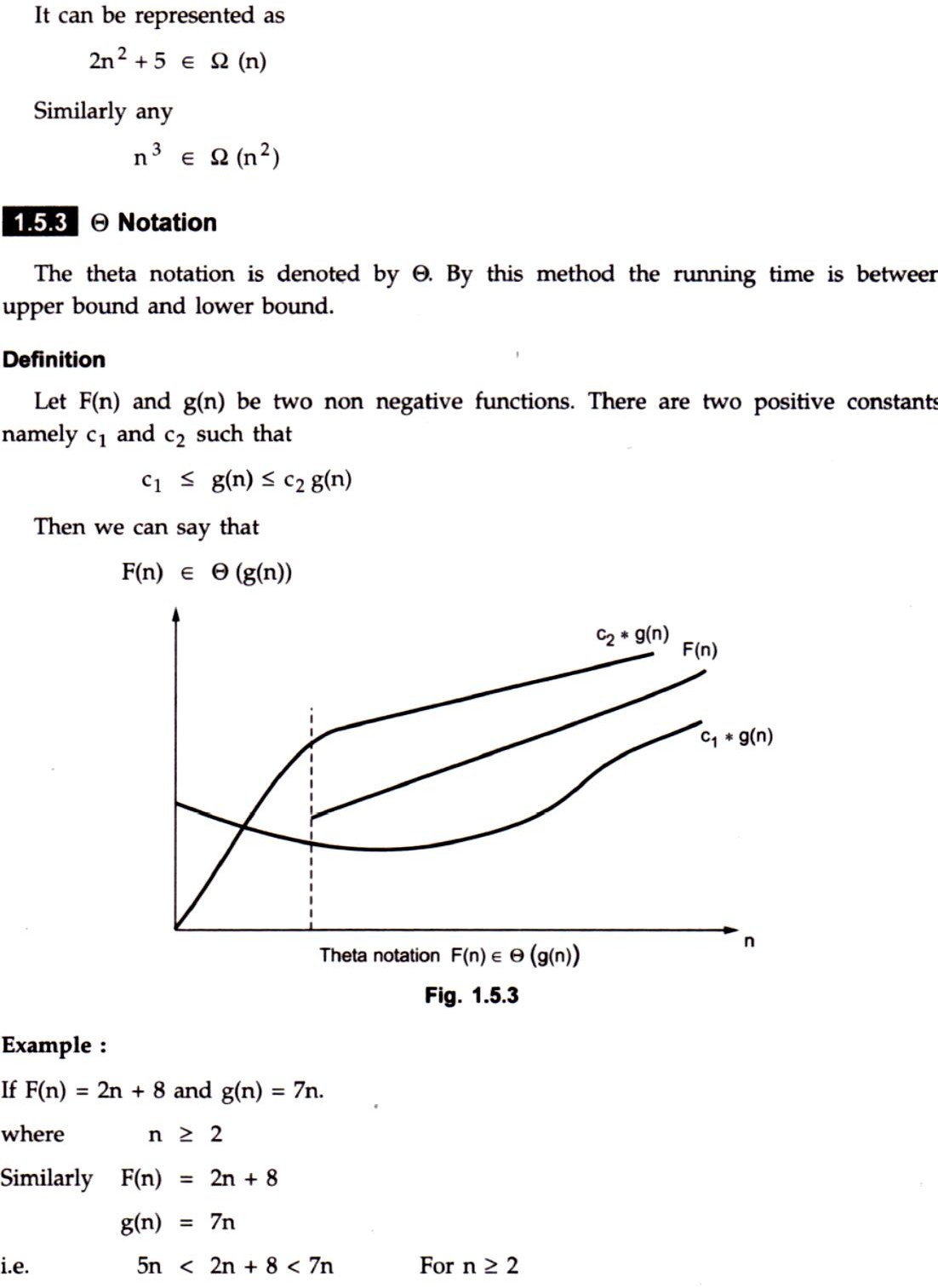


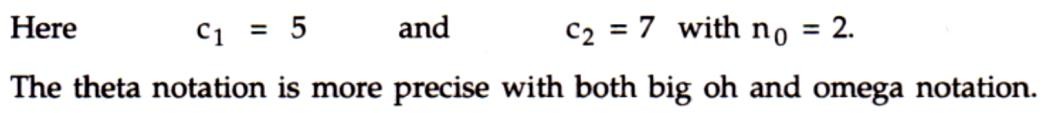
# Omega Notation



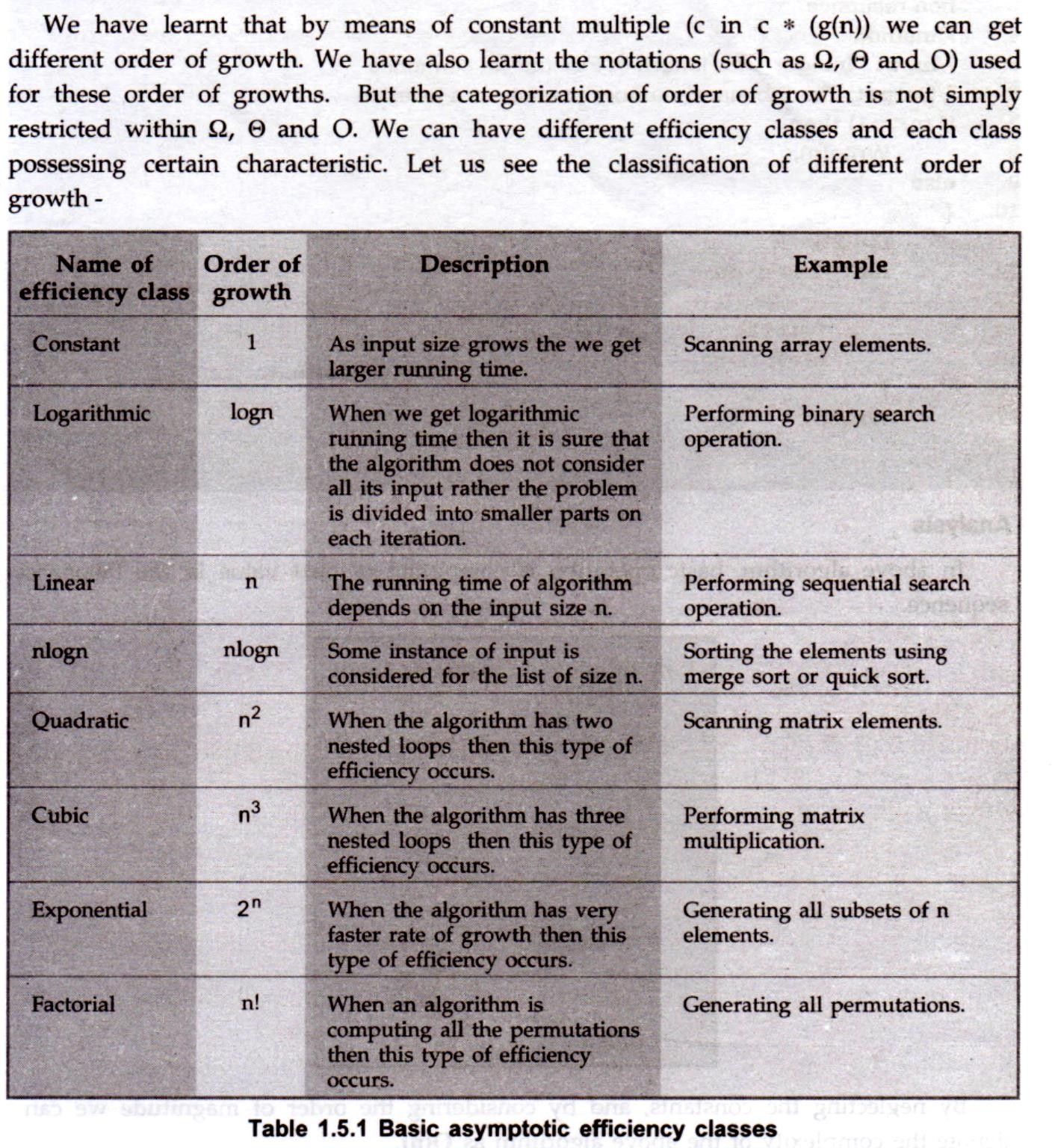








**Basic Efficiency Classes**

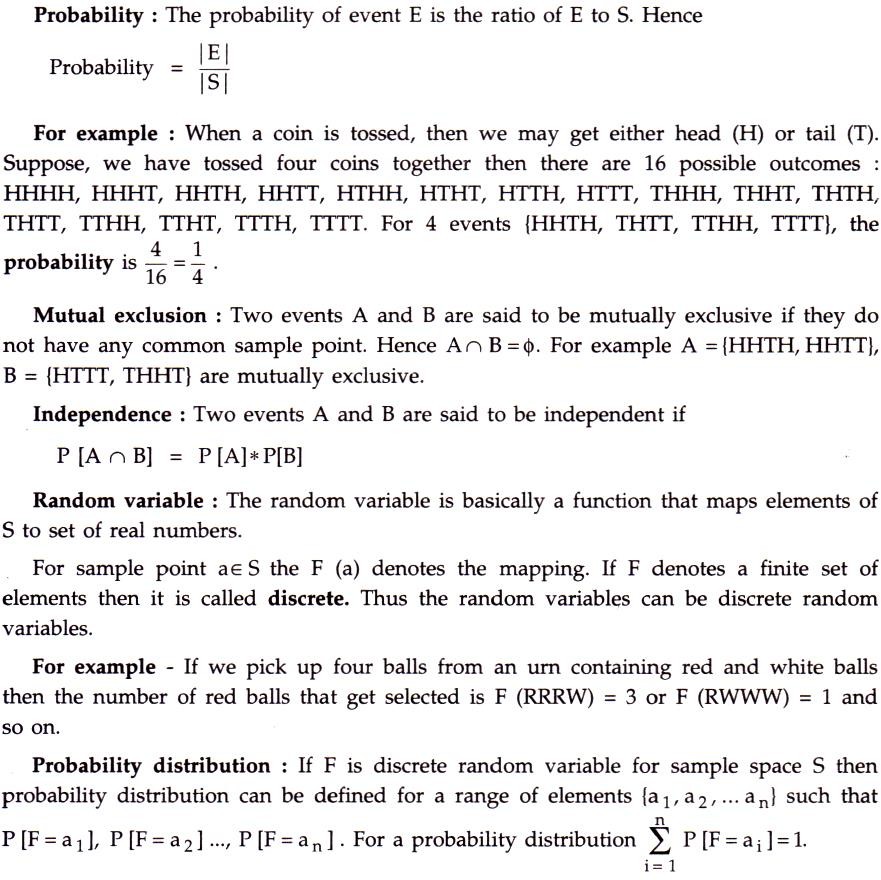


**Probabilistic Analysis**

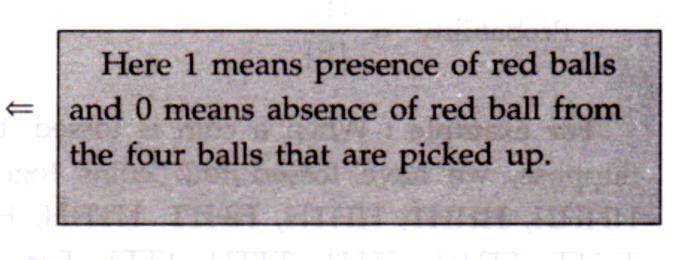
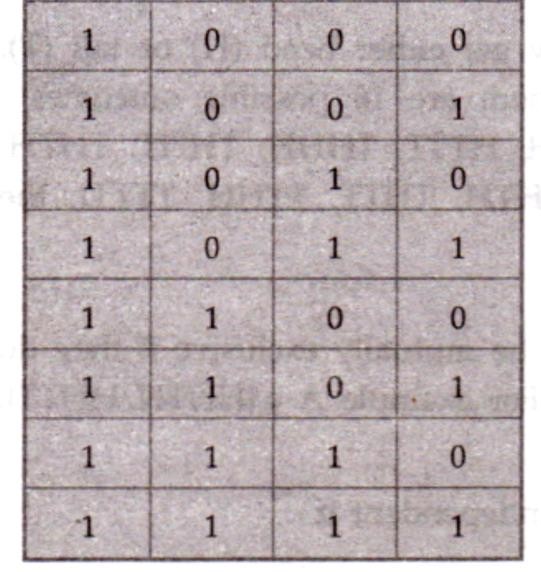
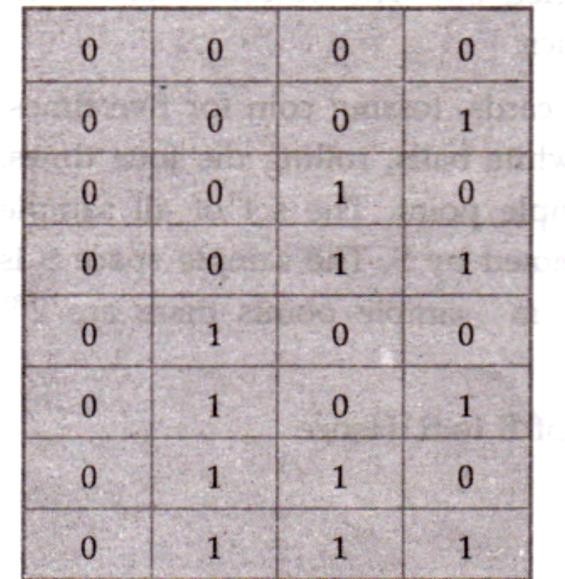
In probability theory various "experiments" are carried out. The outcomes or results of those experiments determine the specific characteristics.

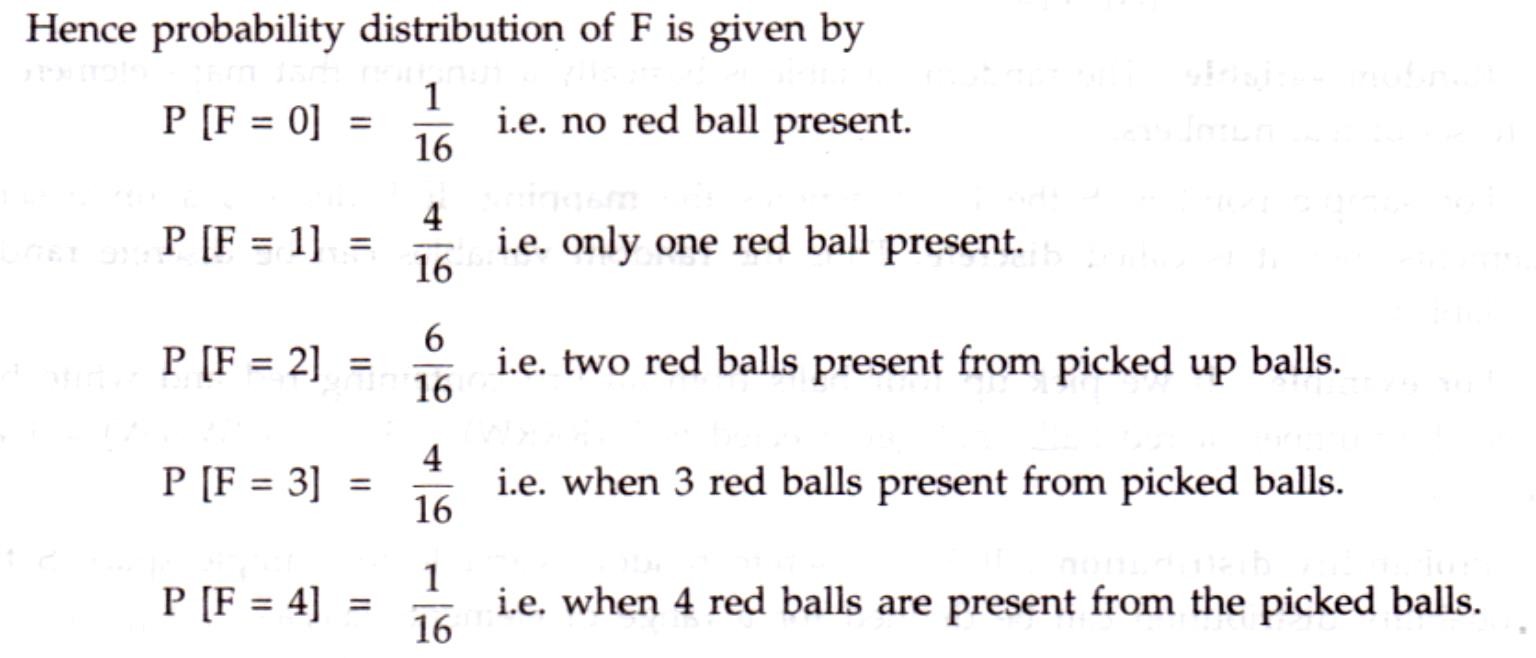
**For example :** Picking up a card from a deck of 52 cards, tossing coin for five times, choosing a red ball from an urn containing red and white balls, rolling die four times. Each possible result of such experiment is called sample point. The set of all sample points is

called **sample space**, The sample space is denoted by S. The sample space S is finite set. An event E occurs from sample space. For **m** sample points there are 2m possible events.



For example if we pick up four balls randomly from an urn containing red and white balls and F is number of red balls, then F can take on five values 0, I, 2,3 and 4 then





**Amortized Analysis**

An amortized analysis means finding average running time per operation over a worst case sequence of operations. An amortized analysis indicates that average cost of a single operation is small if average of sequence of operations is obtained. This is true even if

anyone operation is expensive within the sequence. An amortized analysis guarantees the time per operation over worst case performance.

There is a difference between average case analysis and amortized analysis. In average - case analysis we are averaging over all possible inputs and in amortized analysis we are averaging over a sequence of operations. An amortized analysis assumes worst-case input.

There are 3 commonly used techniques used in amortized analysis -

1. Aggregate analysis
2. Accounting method
3. Potential method

*The amortized cost of n operations is the total cost of the operations divided by n.*

# Finding the Time complexity using Step-Count Method

The step count method is one of the method to analyze the algorithm. In this method, we count number of times one instruction is executing. From that we will try to find the complexity of the algorithm.

Suppose we have one algorithm to perform sequential search. Suppose each instruction will take c1, c2, …. amount of time to execute, then we will try to find out the time complexity of this algorithm

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Number of times** | **Cost** |
| seqSearch(arr, n, key) i := 0  while i < n, do  if arr[i] = key, then break  end if done return i | 1  n+1 n 0/1  1 | c1 c2 c3 c4  c5 |

Now if we add the cost by multiplying the number of times it is executed, (considering the worst case situation), we will get

*Cost=c1+(n+1)c 2+nc3+c 4+c 5 Cost=c1+nc 2+c2+nc 3+c 4+c5 Cost=n(c 2+c3)+c 1+c 4+c5 Cost=n(c 2+c3)+C*

Considering the c1 + c4 + c5 is C, so the final equation is like straight line y = mx + b. So we can say that the function is linear. The complexity will be O(n).

# Frequency Count Method

Basically, frequency count is very simple to understand. It means, the number of times the statement is executed in the program.

Let’s consider some of the program segments

**Program Segment A:**

…

x = x+2

…

**Program Segment B:**

…

for k =1 to n do x = x+2

end

…

**Program Segment C:**

…

for j= 1 to n do for x = 1 to n do x = x+2

end end

…

Now, let us estimate the ***frequency count*** of the statement x = x+2 occurring in the following three program statements.(A, B, C)

The frequency count of the statement in the program segment A is 1.(It gets executed once)

In the program segment B, the frequency count of the statement is n, since the **for** loop in which the statement is embedded executes n(n>=1) times.

In the program segment C, the statement is executed n^2(n>=1) times, since the statement is embedded in a nested **for** loop, executing n times each.

In the program statement C, frequency count will be: (for j= 1 to n) → n+1

(for x =1 to n) → (n+1)n (x = x+2) → n^2

Total frequency count → 3n^2 + 3n +1 Overall statement gets executed n^2 times.

# Tabular Method

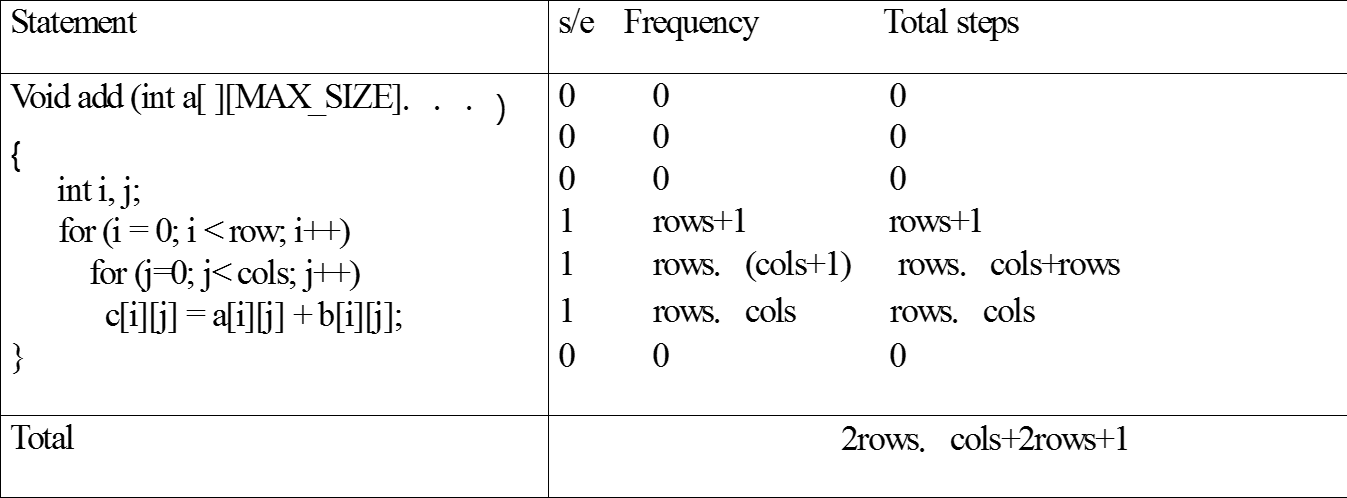
**Example 1: Iterative function to sum a list of numbers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Statement | s/e | Frequency | Total | steps |
| float sum(float list[ ], int n)  {  float tempsum = 0; int i;  for(i=0; i <n; i++) tempsum += list[i];  return tempsum;  } | 0  0  1  0  1  1  1  0 | 0  0  1  0  n+1 n  1  0 | 0  0  1  0  n+1 n  1  0 | |
| Total | 2n+3 | | | |

**Example 2: Step count table for recursive summing function**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Statement | s/e | Frequency | Total | steps |
| float rsum(float list[ ], int n)  {  if (n)  return rsum(list, n-1)+list[n-1]; return list[0];  } | 0  0  1  1  1  0 | 0  0  n+1 n  1  0 | 0  0  n+1 n  1  0 | |
| Total | 2n+2 | | | |

**Example 3: Step count table for matrix addition**



**Best, Average and Worst Case Complexities**

The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its *space complexity* is also important: This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.

There is often a *time-space-tradeoff* involved in a problem, that is, it cannot be solved with few computing time *and* low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

# Best case Time Complexity

The term best-case performance is used in computer science to describe an algorithm's behavior under optimal conditions. For example, the best case for a simple linear search on a list occurs when the desired element is the first element of the list.

Development and choice of algorithms is rarely based on best-case performance: most academic and commercial enterprises are more interested in improving Average-case complexity and worst-case performance. Algorithms may also be trivially modified to have good best-case running time by hard-coding solutions to a finite set of inputs, making the measure almost meaningless.

# Average case Time Complexity

When analyzing algorithms which often take a small time to complete, but periodically require a much larger time, amortized analysis can be used to determine the worst-case running time over a (possibly infinite) series of operations. This amortized worst-case cost can be much closer to the average case cost, while still providing a guaranteed upper limit on the running time.

# Worst case Time Complexity

Worst-case performance analysis and average case performance analysis have some similarities, but in practice usually require different tools and approaches.

Determining what average input means is difficult, and often that average input has properties which make it difficult to characterize mathematically (consider, for instance, algorithms that are designed to operate on strings of text). Similarly, even when a sensible description of a particular "average case" (which will probably only be applicable for some uses of the algorithm) is possible, they tend to result in more difficult analysis of equations.

Worst-case analysis has similar problems: it is typically impossible to determine the exact worst-case scenario. Instead, a scenario is considered such that it is at least as bad as the worst case. For example, when analyzing an algorithm, it may be possible to find the longest possible path through the algorithm (by considering the maximum number of loops, for instance) even if it is not possible to determine the exact input that would generate this path (indeed, such an input may not exist). This gives a safe analysis (the worst case is never underestimated), but one which is pessimistic, since there may be no input that would require this path.

Alternatively, a scenario which is thought to be close to (but not necessarily worse than) the real worst case may be considered. This may lead to an optimistic result, meaning that the analysis may actually underestimate the true worst case.

# Important Questions:

1. What is an algorithm? Explain its characteristics in detail.
2. Write different pseudo code conventions used to represent an algorithm.
3. What is space complexity? Illustrate with an example for fixed and variable part in space complexity.
4. What is an asymptotic notation? Explain different types of asymptotic notations with examples.
5. What is time complexity? Explain the different methods of finding the time complexity with examples.
6. Explain about Best, Average and Worst Case Complexities.