1.1 Notion of an Algorithm

AU : May-14, Marks 8

In this section we will first understand *"What is algorithm?"* and *"When it is required?"*

Definition of 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 definition of algorithm is represented in Fig. 1.1.1.

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.

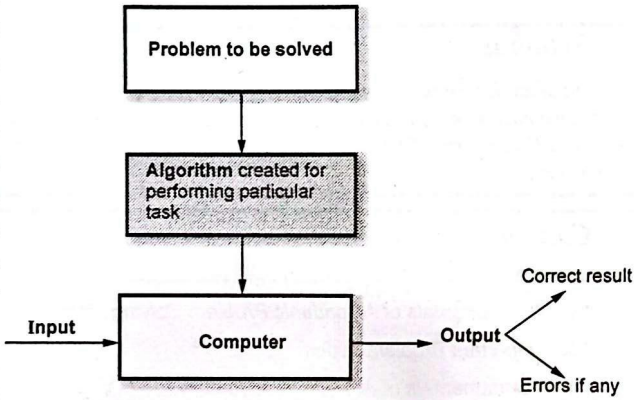


Fig. 1.1.1 Notion of Algorithm

1.1.1 Properties of Algorithm

Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm :

1. **Non-ambiguity :** Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise. The instruction in an algorithm should not denote any conflicting meaning. This property also indicate the *effectiveness* of algorithm.
2. **Range of input :** The range of input should be specified. This is because normally the algorithm is input driven and if the range of the input is not been specified then algorithm can go in an infinite state.

3. **Multiplicity :** The same algorithm can be represented in several different ways. That means we can write in simple English the sequence of instructions or we can write it in the form of pseudo code. Similarly for solving the same problem we can write several different algorithms. For instance : For searching a number from the given list we can use sequential search or a binary search method. Here *"searching"* is a task and use of either a *"sequential search method"* or *"binary search method"* is an algorithm.
4. **Speed :** The algorithms are written using some specific ideas (which is popularly known as logic of algorithm). But such algorithms should be efficient and should produce the output with fast speed.
5. **Finiteness :** The algorithm should be finite. That means after performing required operations it should terminate.

1.1.2 How to Write an Algorithm?

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

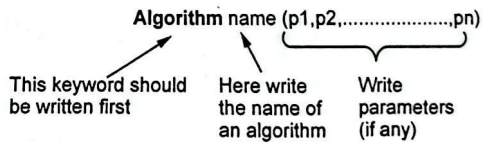
Algorithm Heading
It consists of name of algorithm, problem description, input and output

Algorithm Body
It consists of logical body of the algorithm by making use of various programming constructs and assignment statement.

Fig. 1.1.2 Structure of Algorithm

Let us understand some rules for writing the algorithm.

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



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

```
//Problem Description:
//Input:
//Output:
```

3. Then body of an algorithm is written, in which various programming constructs like if, for, while or some assignment statements may be written.
4. The compound statements should be enclosed within { and } brackets.

5. Single line comments are written using // as beginning of comment.
6. 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.

7. Using assignment operator ← an assignment statement can be given.

For instance :

```
Variable ← expression
```

8. 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 **<, <=, >, >=, =, ≠**
9. 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.
10. The inputting and outputting can be done using **read** and **write**.

For example :

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

11. 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.

12. **while** statement can be written as :

```
while (condition) do
{
    statement 1
    statement 2
        :
        :
    statement n
}
```

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

13. The general form for writing for loop is :

```
for variable ← value1 to valuen do
```



```
{
    statement 1
    statement 2
        :
        :
    statement n
}
```

Here $value_1$ is initialization condition and $value_n$ is a terminating condition. The **step** indicates the increments or decrements in $value_1$ for executing the **for** loop.

Sometime a keyword **step** is used to denote increament or decreament the value of variable for example

```
for i←1 to n step 1
{
        Write (i)
}
```

Here variable i is incremented by 1 at each iteration

14. The **repeat - until** statement can be written as :

```
repeat
    statement 1
    statement 2
        :
        :
    statement n
until (condition)
```

15. 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  1 to 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]
```

1.1.3 Calculating Greatest Common Divisor

The Greatest Common Divisor (GCD) of two non zero numbers a and b is basically the largest integer that divides both a and b evenly i.e. with a remainder of zero. The algorithm for finding GCD is implemented by a famous mathematician ***Euclid Alexandria*** (third century B.C.). Actually one can obtain GCD using three methods -

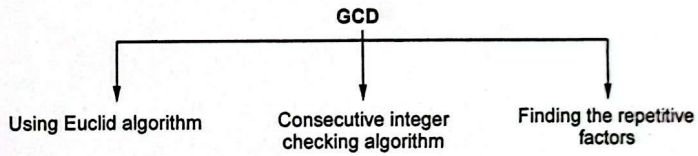


Fig.1.1.3

Let us understand these methods by taking simple example.

1. Finding GCD using Euclid Algorithm -

In this method the GCD is calculated using following function

```
        gcd(a,b)=gcd(b,a mod b)        provided a>b
when we get gcd(a,0)
then
        GCD=a
```

Consider there are two numbers 30 and 18 then we can find GCD as follows -

a	b	Remainder after a/b	
30	18	12	gcd(a, b)=gcd(b, a mod b)
18	12	6	gcd(a, b)=gcd(b, a mod b)
12	6	0	gcd(a, b)=gcd(b, a mod b)
6	0	GCD=6	gcd(a, 0)=GCD=a

The above table itself illustrates the procedure for finding GCD using Euclid's algorithm.

The algorithm in simple steps can be given as -

Step 1 : Divide a by b and assign the value of remainder to variable c.

Step 2 : Then assign the value of b to a and value of c (remainder of a/b) to b.

Step 3 : Repeat the steps 1 and 2 until value of b becomes 0.

Step 4 : If b=0 then return value of a as the GCD value of a and b.

Step 5 : Stop.

We can also write the same algorithm in the form of pseudo code as -

```
Algorithm GCD_Euclid(a,b)
// Problem Description: This algorithm computes the GCD of
//two numbers a and b using Euclid's method.
//Input: two integers a and b.
//Output: GCD value of a and b.
while (b ≠ 0)do
{
 c ← a mod b
 a ← b
 b ← c
}
return a
```

We can write a C function for the above algorithm as follows -

```
int GCD(int a,int b)

{
 while(b!=0)
 {
  c=a%b;
  a=b;
  b=c;
}
return a;
}
```

2. Finding GCD using consecutive integer checking algorithm

In this method while finding the GCD of a and b we first of all find the minimum value of them. Suppose if, value of b is minimum then we start checking the divisibility by each integer which is lesser than or equal to b.

Let us understand this algorithm with the help of some example.

```
Consider a=15 and b=10 then
     t=min(15,10)
```

As 10 is minimum we will set value of t=10 initially. Check whether we get

a mod t=0 as well as *b mod t=0*, if not then decrease *t* by 1 and again with this new *t* value check whether *a mod t = 0* and *b mod t=0*. Thus we go on checking whether *a mod t* and *b mod t* both are resulting 0 or not. Thus we will repeat this process each time by decrementing *t by 1* and performing *a mod t* and *b mod t*. Theses operations are illustrated as follows -

a	b	Description
15 mod 10=5	10 mod 10=0	As *a mod t* is not giving a zero value, t is not a GCD. Set new t = t – 1 t = 10 – 1 t = 9
15 mod 9=6	10 mod 9=1	As *a mod t* and *b mod t* is giving non zero values, t is not GCD new t=t–1 t=9–1 t=8
15 mod 8=7	10 mod 8=2	As *a mod t* and *b mod t* is giving non zero values, t is not GCD new t=t–1 t=8–1 t=7
15 mod 7=1	10 mod 7=3	As *a mod t* and *b mod t* is giving non zero values, t is not GCD new t=t–1 t=7–1 t=6
15 mod 6=3	10 mod 6=4	As *a mod t* and *b mod t* is giving non zero values, t is not GCD new t=t–1 t=6–1 t=5
15 mod 5 = 0	10 mod 5 = 0	As *a mod t* and *b mod t* both are giving zero values, t=5 is a GCD value.

This algorithm is called consecutive integer checking algorithm because after performing *t=min(a,b)* if t is not a GCD value then we check *a mod t* and *b mod t* for the immediate less values of t. Thus consecutive integer values are been checked in this method. Let us now see an algorithm for this method -

Step 1 : Find *min(a, b)*. Assign the minimum value to t.

Step 2 : Then divide *a* by *t*. If remainder is 0 then divide *b* by *t*. If reminder is again 0 then return the value of *t* as a GCD of two numbers *a* and *b*. If any one of the remainder value is non zero then go to step 3.

Step 3 : Decrement value of *t* by 1 and go to step 2.

If any of the input number (either a or b) is 0 then this algorithm will not work correctly. Thus it is important to specify the range of input (one of the property of an algorithm!) while designing an algorithm. The same algorithm can also be written in the form of pseudocode as follows -

```
Algorithm GCD int check(a,b)
// Problem Description: This algorithm computes the GCD of two
//numbers a and b using consecutive integer checking method.
//Input: two integers a and b.
//Output: GCD value of a and b.
t ← min (a,b)
while(t>=1)do
{
 if(a mod t==0 AND b mod t==0) then
   return t
 else
   t ← t-1
}
return 1
```

3. Finding GCD using repetitive factors

We have used this method in schools when we learnt how to find GCD of two numbers.

Consider a=70 and b=28 are the two numbers then

2	70
7	35
5	5
	1

2	28
2	14
7	7
	1

Thus

$$70 = 2 \times 7 \times 5$$

$$28 = 2 \times 2 \times 7$$

$$\therefore \quad GCD = 2 \times 7 = 14$$

The algorithm in simplest form for this method is as given below -

Step 1 : Find the prime factors for *a* and *b*.

Step 2 : Identify the common factors from both the computations made in step 1

Step 3 : Compute the product of the common factors obtained and return the result as the value of GCD. However in this algorithm the first step it is strongly suggested to find the prime factors of given numbers. Hence let us get introduced with the method for finding prime numbers.

Finding prime numbers

This method of finding the prime numbers is suggested by ancient Greece scientist **Sieve of Eratosthenes(c.a. 200 B.C.).** To understand this method let us take an example.

Consider n=23. We will find all the numbers not exceeding 23.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
2	3	#	5	#	7	#	9	#	11	#	13	#	15	#	17	#	19	#	21	#	23
2	3	#	5	#	7	#	#	#	11	#	13	#	#	#	17	#	19	#	#	#	23
2	3	#	5	#	7	#	#	#	11	#	13	#	#	#	17	#	19	#	#	#	23

Here we start by a list of 2 to n numbers. Then we will first find the multiples of 2 and eliminate them i.e we will eliminate 4, 6, 8 and so on. The eliminated numbers are marked by #.

2	3	#	5	#	7	#	9	#	11	#	13	#	15	#	17	#	19	#	21	#	23

Now, we will mark 3 and find multiples of 3. These multiples will be eliminated by marking them as #.

2	3	#	5	#	7	#	#	#	11	#	13	#	#	#	17	#	19	#	#	#	23

Continuing in this fashion we will find the multiples of 5 in the next pass.

2	3	#	5	#	7	#	#	#	11	#	13	#	#	#	17	#	19	#	#	#	23

As $\sqrt{n} = \sqrt{23} \approx 5$. That means the largest number k whose largest number whose multiples can be eliminated is equal to $\sqrt{n}$. Hence we can find the multiplies of $\sqrt{23} \approx 5$. Finally the numbers that are remaining are 2, 3, 5, 7, 11, 13, 17, 19 and 23 which are prime numbers.

Let us write the algorithm for this method.

Step 1 : Consider a list of numbers from 2 to n.

Step 2 : Find the multiples of 2. Here we assume k = 2.

Step 3 : Eliminate these multiplies from the list

Step 4 : Then take the next number say k for finding multiplies of it. If $k > \sqrt{n}$. Then stop by keeping the remaining entries in the list as it is. The numbers that are remaining on the list are prime numbers.

Step 5 : Repeat steps 3 and 4.

The algorithm written in the pseudo code form is also given below -

```
Algorithm prime_Sieve(a[],n)
//Problem Description: This algorithm finds the prime
//numbers that are less than n using Sieve's method.
//Input: Array a and limiting n
//Output: Generates the prime numbers less than n.
for k ← 2;k to n do
 a[k] ← k//store 2 to n numbers in array a
for k ← 2;k to sqrt(n)
{
                if(a[k] ≠ -999)
                {
                    i ← k*k;//finding multiples of k
                    while(i<=n)do
                    {
                      a[i] ← -999//a[i] is eliminated
                      i ← i+k//next multiplies of k
                    }
                }
    }
    for k ← 2 to k<=n do
    { //-999 indicates eliminated entry
        if(a[k] ≠ -999)
         write(a[k]);//displaying the prime numbers
}
```

University Question

1. *What are the features of an efficient algorithm ? Explain.* **AU : May-14, Marks 8**

1.2 Fundamentals of Algorithmic Problem Solving AU : May-14

Let us list **"what are the steps that need to be followed while designing and analysing an algorithm ?"**

1. 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.
- If the given problem is a common type of problem, then **already existing algorithms** as a solution to that problem can be used.
- After applying such an existing algorithm it is necessary to find its strength and weakness (For example, efficiency, memory utilization).

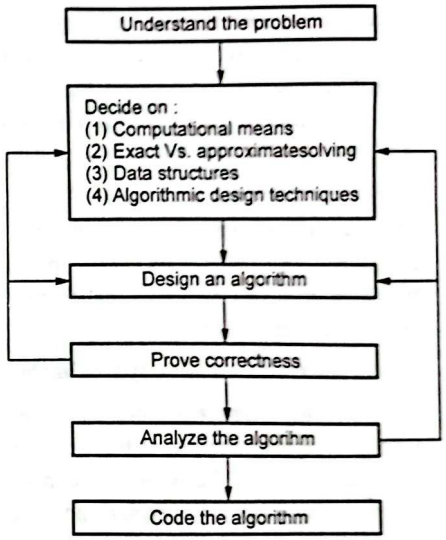


Fig. 1.2.1 Algorithmic analysis and design process

- However, it is very rare to have such a readymade algorithm. 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.**

2. Decision making

- After finding the required input set for the given problem we have to analyze the input.
- We 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.**

a. Capabilities of computational devices

It is necessary to know the computational capabilities of devices on which the algorithm will be running. Globally we can classify an algorithm from execution point of view as **sequential algorithm** and **parallel algorithm.** The sequential algorithm specifically runs on the machine in which the instructions are executed one after another. Such a machine is called as Random Access Machine (RAM). And the parallel algorithms are run on the machine in which the instructions are executed in parallel.

There are certain complex problems which require huge amount of memory or the problems for which execution time is an important factor. For solving such problems it is essential to have proper choice of a **computational device** which is **space and time efficient.**

b. Choice for either exact or approximate problem solving method

The next important decision is to decide whether the problem is to be solved exactly or approximately. If the problem needs to be solved correctly then we need **exact algorithm.** Otherwise if the problem is so complex that we won't get the exact solution then in that situation we need to choose **approximation algorithm.** The typical example of approximation algorithm is travelling salesperson problem.

c. Data structures

Data structure and algorithm work together and these are interdependent. Hence choice of proper data structure is required before designing the actual algorithm. The implementation of algorithm (program) is possible with the help of algorithm and data structure.

d. Algorithmic design techniques

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 Techniques

- ***Brute Force :*** This is a straightforward technique with naive approach.
- ***Divide-and-Conquer :*** The problem is divided into smaller instances.
- ***Decrease-and-Conquer :*** The instance size is decreased to solve the problem.
- ***Transform-and-Conquer :*** The instance is modified and then solved.
- ***Dynamic Programming :*** The results of smaller, reoccurring instances are obtained to solve the problem.

3. Design an algorithm

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

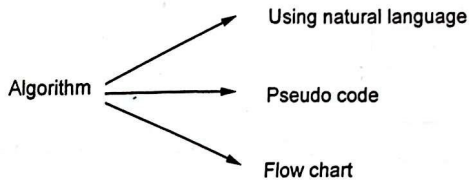


Fig. 1.2.2

Natural language

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 there by we get brief specification.

For example : Write an algorithm to perform addition of two numbers.

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

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

```
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.

Flow chart

Another way of representing the algorithm is by **flow chart.** Flow chart is a graphical representation of an algorithm. Typical symbols used in flow chart are -

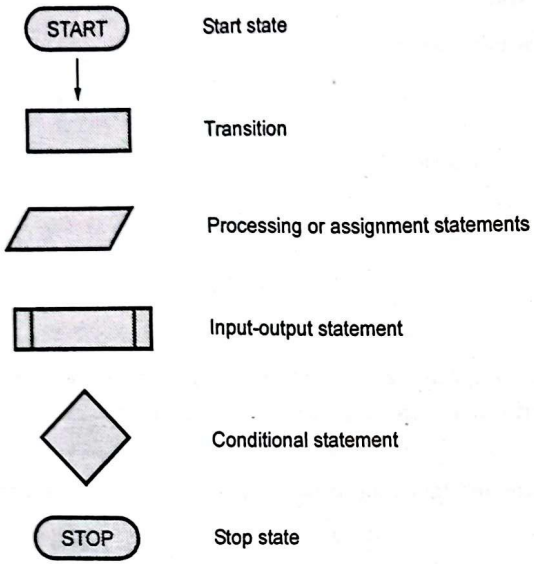


Fig. 1.2.3 Symbols used in flowchart

For example

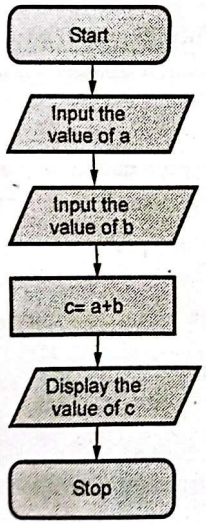


Fig. 1.2.4 Flow chart for addition of two number

4. Prove correctness

- 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.** But to show that an algorithm works incorrectly we have to show that **at least for one instance** of valid input the algorithm gives wrong result.

5. Analysis of algorithm

While analyzing an algorithm we should consider following factors -

1. **Time complexity** of an algorithm means the amount of time taken by an algorithm to run. By computing time complexity we come to know whether the algorithm is slow or fast.
2. **Space complexity** of an algorithm means the amount of space (memory) taken by an algorithm. By computing space complexity we can analyze whether an algorithm requires more or less space.
3. **Simplicity** of an algorithm means generating sequence of instructions which are easy to understand. This is an important characteristic of an algorithm because simple algorithms can be understood quickly and one can then write simpler programs for such algorithms. While simplifying an algorithm we have to compute any predefined computations or some complex mathematical derivation. Finding out bugs from algorithms or debugging the program becomes easy when an algorithm is simple.
4. **Generality** sometimes it becomes easier to design an algorithm in more general way rather than designing it for particular set of input. Hence we should write general algorithms always. For example, designing an algorithm for finding GCD of any two numbers is more appealing than that of particular two values. But sometimes it is not at all required to design a generalized algorithm. For example, an algorithm for finding roots of quadratic equations cannot be designed to handle a polynomial of arbitrary degree.
5. **Range of inputs** comes in picture when we execute an algorithm. The design of an algorithm should be such that it should handle the range of input which is most natural to corresponding problem.

Analysis of algorithm means checking the characteristics such as : Time complexity, space complexity, simplicity, generality and range of input. If these factors are not satisfactory then we must redesign the algorithm.

6. Code the 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.

University Question

1. *What is space complexity ? With an example, explain the components of fixed and variable part in space complexity.* **AU : May-14, Marks 8**

1.3 Important Problem Types

There is large number of computing problems and some of them can be classified as

1. Sorting
2. Searching
3. Numerical problems
4. Geometric problems
5. Combinatorial problems
6. Graph problems
7. String processing problems

These important problem types can be discussed in detail as follows

1. Sorting

Sorting means arranging the elements in increasing order or in decreasing order (also called as **ascending order** or **descending order** respectively). The sorting can be done on numbers, characters (alphabets), strings or employees record. For sorting any record we need to choose certain piece of information based on which sorting can be done. For instance : For keeping the employees record in sorting order we will arrange the employees record as per employee ID. Similarly one can arrange the library books according to title of the books. This piece of information which is required to sort the records is called **key**. The important property of this key is that it should be unique.

What is the need for sorting ?

The usefulness of sorting is that we can search any desired element efficiently. That is why the telephone directory, dictionary or any database is always in sorted order.

Properties of Sorting

In computer science there are numerous sorting algorithms that are available. All can be analyzed by two important properties : Stable property and in-place property.

The algorithm is said to be stable if it preserves the relative ordering of items with equal values. For example : Consider a following list -

Roll Number	Name
10	Padma
20	Jayashri
30	Sarika
40	Sarika
50	Poonam

As it is clear that, the above list is sorted based on the field *-Roll Number*. If we sort the list based on a second field then the list is -

Roll Number	Name
20	Jayashri
10	Padma
50	Poonam
30	Sarika
40	Sarika

Note that we have two records <30, Sarika> and <40, Sarika> in which the second field is same. But while sorting the records the order of records is preserved. If the algorithm is not stable then it is possible that we <40, Sarika> may appear before <30, Sarika>

An algorithm is said to be **in-place** if it does not require extra memory while sorting the list. There are some other algorithms which require extra memory while sorting the list of elements ; then such algorithms are not **in-place** algorithms.

2. Searching

Searching is an activity by which we can find out the desired element from the list. The element which is to be searched is called **search key**. There are many searching algorithms such as sequential search, binary search, Fibonacci search and many more.

How to choose best searching algorithm ?

One can not declare that a particular algorithm is best searching algorithm because some algorithms work faster, but then they may require more memory. Some algorithms work better if the list is almost sorted. Thus efficiency of algorithm is varying at varying situations.

Searching in dynamic set of elements

There may be a set of elements in which repeated addition or deletion of elements occur. In such a situation searching an element is difficult. To handle such lists supporting data structures and algorithms are needed to make the list balanced (organized).

3. Numerical Problems

The numerical problems are based on mathematical equations, systems of equations, computing definite integrals, evaluating functions and so on. These mathematical problems are generally solved by approximate algorithms. These algorithms require manipulating of the real numbers; hence we may get wrong output many times.

4. Geometric Problems

The geometric problems is one type of problem solving area in which various operations can be performed on geometric objects such as points, line, polygon.

The geometric problems are solved mainly in applications to computer graphics or in robotics.

5. Combinatorial Problems

The combinatorial problems are related to the problems like computing permutations and combinations. The combinatorial problems are most difficult problems in computing area because of following causes

- As problem size grows the combinatorial objects grow rapidly and reach to a huge value.
- There is no algorithm available which can solve these problems in finite amount of time.
- Many of these problems fall in the category of unsolvable problems.

However there are certain problems which are solvable.

6. Graph Problems

Graph is a collection of vertices and edges. The graph problems involve graph traversal algorithms, shortest path algorithms and topological sorting and so on. Some graph problems are very hard to solve. For example travelling salesman problem, graph coloring problems.

7. String Processing Problems

String is a collection of characters. In computer science the typical string processing algorithm is **pattern matching** algorithm. In these algorithms particular word is searched from the text. The algorithms lying in this category are simple to implement.

1.4 Fundamentals of the Analysis of Algorithmic Efficiency

- Efficiency of an algorithm can be in terms of **time or space**. Thus, checking whether the algorithm is efficient or not means analyzing the algorithm.
- There is a systematic approach that has to be applied for analyzing any given algorithm. This systematic approach is modelled by a framework called as **analysis framework.**

Let us understand the analysis framework in detail.

1.5 Analysis Framework

AU : May-10, Marks 8

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.(See Fig. 1.5.1 on next page).

1.5.1 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**

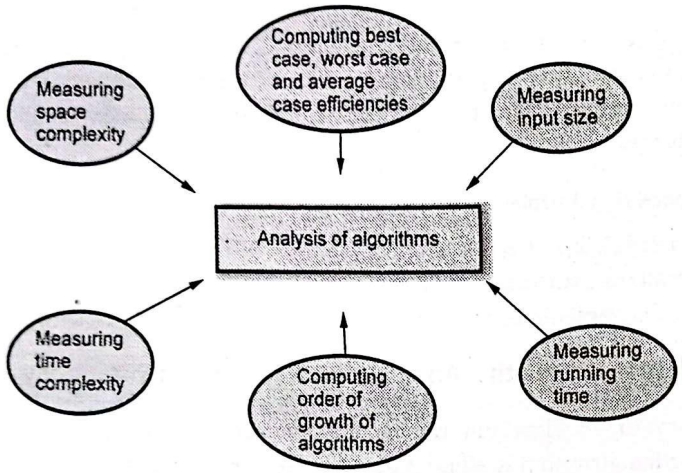


Fig. 1.5.1 Analysis of algorithms

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 Example 1 is

$S(p)=C \qquad \Theta(Sp) = 0$

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 -

$S(p) \geq (n + 3)$

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

Example 3 :

```
Algorithm Add (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 Add (x,n–1)+x[n]
```

The space requirement is -

$S(p) \geq 3(n + 1)$

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 atleast 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).

1.5.2 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.

For Example

If we write a code for calculating sum of n numbers in an array then we can find its time complexity using frequency count. This frequency count denotes how many times the particular statement is executed.

```
for(i=0; i<n; i++)
{
    sum = sum+a[i];
}
```

Statement	Frequency Count
i = 0	1
i < n	This statement executes for (n + 1) times. When conditions is true i.e. when i < n is true, the execution happens to be n times, and the statement executes once more when i < n is false.
i ++	n times
Sum = Sum + a [i]	n times
Total	**3n + 2**

Thus we get frequency count to be $3n+2$. The time complexity is normally denoted in terms of Oh notation (O). Hence if we neglect the constants then we get the time complexity to be **O (n)**

Another example

Let us consider another example. The following code is typically used to perform matrix addition.

```
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
    {
        c[i][j] = a[i][j]+b[i][j]
    }
}
```

The frequency count can be computed as follows.

- i = 0 executes once ∴ Frequency count = 1
- i < n executes for n +1 times.
- i + + executes for n times.
- j = 0 executes for

$$\underset{\text{for outer loop}}{n} \times \underset{\text{for initialization of j}}{(1)} = n \text{ times}$$

- j < n executes for

$$\underset{\text{execution of outer i loop}}{n} \times \underbrace{(n+1)}_{\text{execution of inner j loop}} = n^2 + n \text{ times}$$

- j + + executes for

$$\underset{\text{execution of outer i loop}}{n} \times \underset{\text{execution of inner j loop}}{(n)} = n^2 \text{ times}$$

- c [i] [j] = a [i] [j] + b [i] [j] executes for

$$\underset{\text{execution of outer i loop}}{n} \times \underset{\text{execution of inner j loop}}{(n)} = n^2 \text{ times}$$

Hence frequency count is $= 3n^2 + 4n + 2$

If the constants are neglected then the time complexity will be $O(n^2)$.

1.5.3 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.

1.5.4 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.

Problem statement	Input size	Basic operation
Searching a key element from the list of n elements.	List of n elements.	Comparison of key with every element of list.
Performing matrix multiplication.	The two matrices with order n × n.	Actual multiplication of the elements in the matrices.
Computing GCD of two numbers.	Two numbers.	Division.

Table 1.5.1 Basic operations from input

- 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.

$$T(n) \approx c_{op}\, C(n)$$

- Running time of basic operation
- Time taken by the basic operation to execute
- Number of times the operation needs to be executed

1.5.5 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.

n	log n	n log n	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Table 1.5.2 Order of growth

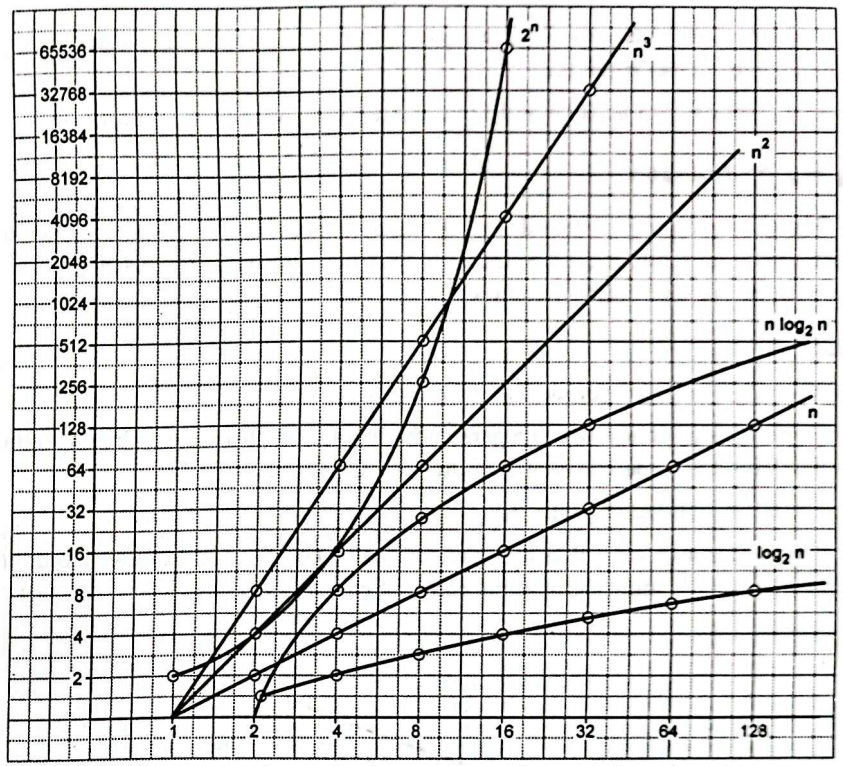


Fig. 1.5.2 Rate of growth of common computing time function

From the above drawn graph Fig. 1.5.2 it is clear that the logarithmic function is the slowest growing function. And the exponential function 2^n 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 $2^{16} = 65536$.

We have plotted the graph for these values, as shown in Fig. 1.5.2.

University Question

1. *Explain how time complexity is calculated. Give an example.* **AU : May-10, Marks 8**

1.6 Asymptotic Notations and its Properties

AU : May-10,11,12,13,15,16,17,18, Dec.-10,17, Marks 16

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, Θ and O used are called **asymptotic notions.**

1.6.1 Big oh Notation

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 n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$f(n) \leq c * g(n)$$

then f(n) is big oh of g(n). It is also denoted as $f(n) \in O(g(n))$. In other words f(n) is less than g(n) if g(n) is multiple of some constant c.

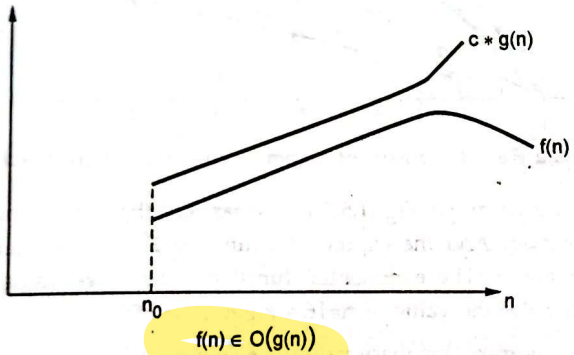


Fig. 1.6.1

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

$$f(n) = 2n+ 2$$
$$= 2(1) + 2$$
$$f(n) = 4$$

and $$g(n) = n^2$$

$$= (1)^2$$
$$g(n) = 1$$

i.e. $f(n) > g(n)$

If $n = 2$ then,

$$f(n) = 2(2) + 2$$
$$= 6$$
$$g(n) = (2)^2$$
$$g(n) = 4$$

i.e. $f(n) > g(n)$

If $n = 3$ then,

$$f(n) = 2(3) + 2$$
$$= 8$$
$$g(n) = (3)^2$$
$$g(n) = 9$$

i.e. $f(n) < g(n)$ is true.

Hence we can conclude that for $n > 2$, we obtain

$$f(n) < g(n)$$

Thus always upper bound of existing time is obtained by big oh notation.

1.6.2 Omega Notation

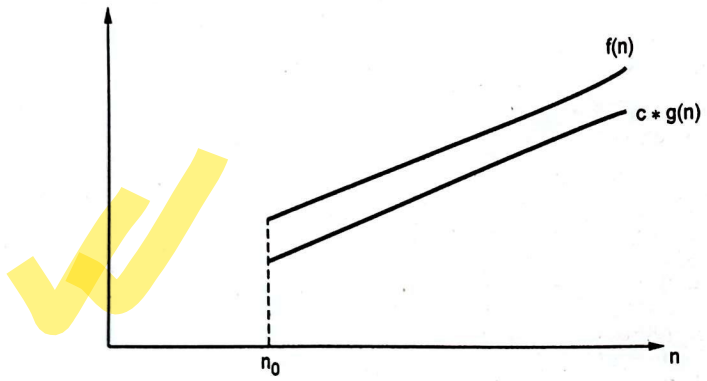
Omega notation is denoted by 'Ω'. This notation is used to represent the **lower bound** of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition

A function f(n) is said to be in Ω (g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that

$$f(n) \geq c * g(n) \qquad \text{For all } n \geq n_0$$

It is denoted as $f(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.



$f(n) \in \Omega(g(n))$

Fig. 1.6.2

Example :

Consider $f(n) = 2n^2 + 5$ and $g(n) = 7n$

Then if $n = 0$

$f(n) = 2(0)^2 + 5$

$= 5$

$g(n) = 7\ (0)$

$= 0$ i.e. $f(n) > g(n)$

But if $n = 1$

$f(n) = 2(1)^2 + 5$

$= 7$

$g(n) = 7(1)$

$= 7$ i.e. $f(n) = g(n)$

If n = 3 then,

$f(n) = 2(3)^2 + 5$

$= 18 + 5$

$= 23$

$g(n) = 7(3)$

$= 21$

i.e. $f(n) > g(n)$

Thus for $n > 3$ we get $f(n) > c * g(n)$.

It can be represented as

$2n^2 + 5 \in \Omega(n)$

Similarly any

$n^3 \in \Omega(n)^2$

1.6.3 Θ Notation

The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

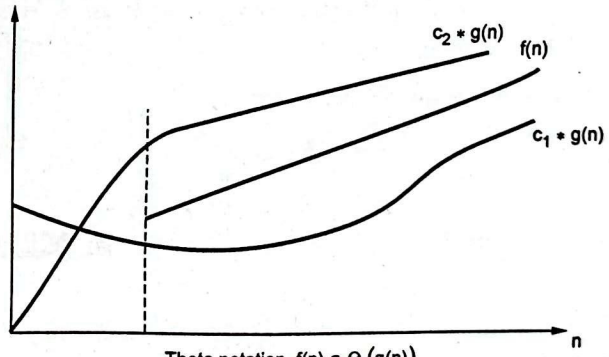
Definition

Let f(n) and g(n) be two non negative functions. There are two positive constants namely c_1 and c_2 such that

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

Then we can say that

$f(n) \in \Theta(g(n))$



Theta notation $f(n) \in \Theta(g(n))$

Fig. 1.6.3

Example :

If $f(n) = 2n + 8$ and $g(n) = 7n$.

where $n \geq 2$

Similarly $f(n) = 2n + 8$

$g(n) = 7n$

i.e. $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$.

The theta notation is more precise with both big oh and omega notation.

1.6.4 Properties of Order of Growth

1. If $f_1(n)$ is order of $g_1(n)$ and $f_2(n)$ is order of $g_2(n)$, then
   $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))$.
2. Polynomials of degree $m \in \Theta(n^m)$.

   That means maximum degree is considered from the polynomial.

   For example : $a_1 n^3 + a_2 n^2 + a_3 n + c$ has the order of growth $\Theta(n^3)$.
3. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$.
4. Exponential functions a^n have different orders of growth for different values of a.

Key Point *i) $O(g(n))$ is a class of functions $F(n)$ that grows less fast than $g(n)$, that means $F(n)$ possess the time complexity which is always lesser than the time complexities that $g(n)$ have.*

ii) $\Theta(g(n))$ is a class of functions $F(n)$ that grows at same rate as $g(n)$.

iii) $\Omega(g(n))$ is a class of functions $F(n)$ that grows faster than or atleast as fast as $g(n)$. That means $F(n)$ is greater than $\Omega(g(n))$.

Example 1.6.1 *Show the following equalities are correct.*

i) $5n^2 - 6n = \Theta(n^2)$

ii) $n! = O(n^n)$

iii) $n^3 + 10^6 n^2 = \Theta(n^3)$

iv) $2n^2 2^n + n \log n = \Theta(n^2 2^n)$ **AU : May-13, Marks 16**

Solution :

i) $5n^2 - 6n = \Theta(n^2)$

The given function $f(n) \in \Theta(g(n))$ only when $f(n)$ can be represented as

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$f(n) = 5n^2 - 6n$. This can be denoted as $n^2 \leq 5n^2 - 6n \leq 5n^2$ for all $n \geq 2$.

Where $g(n) = n^2$, $c_1 = 1$ and $c_2 = 5$. Hence given equality holds true.

ii) $n! = O(n^n)$

The given function $f(n) \in O(g(n))$ only

when $f(n)$ can be represented by

$f(n) \leq c * g(n)$.

Here $f(n) = n!$ and $g(n) = n^n$

The equation can be represented as,

$n \cdot (n-1) \cdot (n-2) \ldots 1 \leq n \cdot n \cdot n \cdot \ldots n$ for all $n \geq 1$

This shows that $f(n) \leq c * g(n)$ where $c = 1$.

Hence given equality holds true.

iii) $n^3 + 10^6 n^2 = \Theta(n^3)$

The given function is $f(n) \in \Theta(g(n))$, only

when $f(n)$ can be represented as

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$.

$f(n) = n^3 + 10^6 n^2$, $g(n) = n^3$.

This can be denoted as -

$n^3 \leq n^3 + 10^6 n^2 \leq 10^7 n^3$ for all $n \geq 1$

and $c_1 = 1$, $c_2 = 10^7$

Hence given equality is true.

iv) $2n^2 2^n + n \log n = \Theta(n^2 2^n)$

The given function $f(n) \in \Theta(g(n))$, only

when $f(n)$ can be represented as

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$.

$f(n) = 2n^2 2^n + n \log n$. This can also be denoted as -

$c_1 n^2 2^n \leq 2n^2 2^n + n \log n \leq c_2 n^2 2^n$

$$c_1 \leq 2 + \frac{n \log n}{n^2 2^n} \leq c_2$$

where for all $n \geq 1$, $0 \leq \frac{n \log n}{n^2 2^n} \leq 1$

with $c_1 = 2$, $c_2 = 3$ and $n_0 = 1$

Hence given equality is true.

Example 1.6.2 *State whether following equality is correct or wrong ?*

$5n^3 + 4n = \Omega(n^2)$

Solution : To prove that given function $f(n) \in \Omega(g(n))$, only when $f(n)$ can be represented as $f(n) \geq c * g(n)$ where $n \geq n_0$.

Here, assume , $f(n) = 5n^3 + 4n$ and $g(n) = n^2$.

$5n^3 + 4n \geq n^2$ when $c = 1, n_0 = 0$

for all $n \geq n_0$. This shows that given equality is true.

Example 1.6.3 *State whether following equality is correct or wrong ?*

$10n^3 + n \log n + 9 = O(n^2)$.

Solution : We assume that, we have obtained the values of $c\ n_0$ in such a way that $n >= n_0$ and we have $10n^3 + n \log n + 9 < cn^2$. Now Let, $m = \max(n_0, \text{ceiling } (c/10))$, then

We have,

$10m^3 + m \log m + 9 < cm^2$

$= 10m^3 + m \log m + 9 - cm^2 < 0$

$= m^2(10m - c) + m \log m + 9 < 0$

But $10m - c > 0$, hence

$m^2(10m - c) + m \log m + 9 < 0$ is not possible.

That means there are no such values of c and n_0 such that $n \geq n_0$ and

$10n^3 + n \log n + 9 < c\,n^2$. Hence given equality is wrong.

Example 1.6.4 *Derive a loose bound on the following equation :*

$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$ **AU : May-15, Marks 8**

Solution : Let, f(n) and g(n) are two non-negative functions.

Let c be some constant.

The equation

$f(n) \leq c * g(n)$

then $f(n) \in O(g(n)$ with tight bound.

But if $f(n) < c * g(n)$

then $f(n) \in O(g(n)$ with loose bound.

Consider the function

$f(x) = 35x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$

and $g(x) = x^8$

If $x = 1$, then

$f(x) = 35(1)^8 - 22(1)^7 + 14(1)^5 - 2(1)^4 - 4(1)^2 + 1 - 15$

$f(x) = 7$

$g(x) = x^8 = (1)^8$

If we assume $c = 35$ then

We will always get

$f(x) < g(x)$ for $x \geq 1$

1.6.5 Basic Efficiency Classes

We can have different efficiency classes and each class possessing certain characteristic. Let us see the classification of different order of growth -

Name of efficiency class	Order of growth	Description	Example
Constant	1	As input size grows the we get larger running time.	Scanning array elements.
Logarithmic	log n	When we get logarithmic running time then it is sure that the algorithm does not consider all its input rather the problem is divided into smaller parts on each iteration.	Performing binary search operation.
Linear	n	The running time of algorithm depends on the input size n.	Performing sequential search operation.
n log n	n log n	Some instance of input is considered for the list of size n.	Sorting the elements using merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cubic	n^3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs.	Generating all subsets of n elements.
Factorial	n!	When an algorithm is computing all the permutations then this type of efficiency occurs.	Generating all permutations.

Table 1.6.1 Basic asymptotic efficiency classes

University Questions

1. *Elaborate on asymptotic notations with examples.* **AU : May-10,11, Marks 8**
2. *Prove that for any two functions f(n) and g(n), we have f(n) = Θ (g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).* **AU : Dec.-10 Marks 6**
3. *Discuss in detail all the asymptotic notations with examples.* **AU : May-12, Marks 16**
4. *Give the definition and graphical representation of O-notation.* **AU : May-16, Marks 8**
5. *Explain briefly Big oh Notation, Omega Notation and Theta Notations. Give examples.* **AU : May-17, Marks 13**
6. *What are the rules of manipulate Big- oh expressions and about the typical growth rates of algorithm.* **AU : Dec.-17, Marks 13**
7. *Define Big O notation, Big Omega and Big Theta Notation. Depict the same graphically and explain.* **AU : May-18, Marks**

1.7 Properties of Big oh

AU : May-11, Dec.-14, Marks 16

Following are some important properties of big oh notations -

1. If there are two functions $f_1(n)$ and $f_2(n)$ such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = \max(O(g_1(n)), O(g_2(n)))$.
2. If there are two functions $f_1(n)$ and $f_2(n)$ such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$.
3. If there exists a function f_1 such that $f_1 = f_2 * c$ where c is the constant then, f_1 and f_2 are equivalent. That means $O(f_1 + f_2) = O(f_1) = O(f_2)$.
4. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.
5. In a polynomial the highest power term dominates other terms.
   For example if we obtain $3n^3 + 2n^2 + 10$ then its time complexity is $O(n^3)$.
6. Any constant value leads to O(1) time complexity. That is if $f(n) = c$ then it $\in O(1)$ time complexity.
7. If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in O(g(n))$ but $f(n) \notin \theta(g(n))$.

Example with Solution

Example 1.7.1 *Is Big Oh notation transitive ? Discuss with an example.* **AU : May-11, Marks 4**

Solution : Yes. The Big Oh notation is transitive.

i.e. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then

$f(n) = O(h(n))$.

Consider $f(n) = n^3$

$g(n) = 4n^3 + 4n$

$h(n) = 105n^3 + 4n^2 + 6n$

Then if $f(n) = O(g(n))$ is true then

$g(n) = O(h(n))$ is also true

The n^3, $4n^3 + 4n$ and $105n^3 + 4n^2 + 6n = O(n^3)$

University Question

1. *Discuss the properties of big-oh notation.* **AU : Dec.-14, Marks 16**

1.8 Recurrence Equation

AU : May-10,12, Dec.-10,11,12,14,15,16, Marks 16

The recurrence equation is an equation that defines a sequence recursively. It is normally in following form -

$$T(n) = T(n-1) + n \quad \text{for } n > 0 \quad \ldots (1)$$

$$T(0) = 0 \quad \ldots (2)$$

Here equation (1) is called **recurrence relation** and equation (2) is called **initial condition.** The recurrence equation can have infinite number of sequences. The **general solution** to the recursive function specifies some formula.

For example : Consider a recurrence relation

$$T(n) = 2T(n-1) + 1 \quad \text{for } n > 1$$

$$T(1) = 1$$

Then by solving this recurrence relation we get $f(n) = 2^n - 1$. When n = 1, 2, 3, and 4.

1.8.1 Solving Recurrence Equations

The recurrence relation can be solved by following methods -

1. Substitution method 2. Tree method 3. Master's method.

Let us discuss these methods with suitable examples -

1) Substitution Method

The substitution method is a kind of method in which a **guess** for the solution is made.

There are two types of substitution -

- Forward substitution
- Backward substitution.

Forward Substitution Method - This method makes use of an initial condition in the initial term and value for the next term is generated. This process is continued until some formula is guessed. Thus in this kind of substitution method, we use recurrence equations to generate the few terms.

For example :

Consider a recurrence relation

$$T(n) = T(n-1) + n$$

with initial condition T(0) = 0.

Let,

$$T(n) = T(n-1) + n \quad \ldots (3)$$

If n = 1 then

$$T(1) = T(0) + 1$$

$$= 0 + 1 \quad \because \text{Initial condition}$$

$$\therefore \quad \mathbf{T(1) = 1} \quad \ldots (4)$$

If n = 2, then

$$T(2) = T(1) + 2$$

$$= 1 + 2 \quad \because \text{Equation 4}$$

$$\therefore \quad \mathbf{T(2) = 3} \quad \ldots (5)$$

If n = 3 then

$$T(3) = T(2) + 3$$

$$= 3 + 3 \quad \because \text{Equation 5}$$

$$\therefore \quad \mathbf{T(3) = 6} \quad \ldots (1.9.6)$$

By observing above generated equations we can derive a formula

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

We can also denote T(n) in terms of big oh notation as follows -

$$T(n) = O(n^2)$$

But in practice, it is difficult to guess the pattern from forward substitution. Hence this method is not very often used.

Backward Substitution Method -

In this method backward values are substituted recursively in order to derive some formula.

For example -

Consider, a recurrence relation

$$T(n) = T(n-1) + n \quad \ldots (7)$$

With initial condition T(0) = 0

$$T(n-1) = T(n-1-1) + (n-1) \quad \ldots (8)$$

Putting equation (8) in equation (7) we get,

$$T(n) = T(n-2) + (n-1) + n \quad \ldots (9)$$

Let

$$T(n-2) = T(n-2-1) + (n-2) \quad \ldots (10)$$

Putting equation (10) in equation (9) we get,

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

$$\vdots$$

$$= T(n-k) + (n-k+1) + (n-k+2) + \ldots + n$$

If k = n then

$$T(n) = T(0) + 1 + 2 + \ldots n$$

$$T(n) = 0 + 1 + 2 + \ldots + n \quad \because T(0) = 0$$

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

Again we can denote T(n) in terms of big oh notation as

$$\mathbf{T(n) \in O(n^2).}$$

Some Examples on Recurrence Relation

Example 1.8.1 *Solve the following recurrence relation T(n) = T(n − 1) + 1 with T(0) = 0 as initial condition. Also find big oh notation.*

Solution : Let,

$$T(n) = T(n-1) + 1$$

By backward substitution,

$$T(n-1) = T(n-2) + 1$$

$$\therefore \quad T(n) = T(n-1) + 1$$

$$= (T(n-2) + 1) + 1$$

$$\mathbf{T(n) = T(n-2) + 2}$$

Again $T(n-2) = T(n-2-1)+1$

$= T(n-3)+1$

$\therefore \quad T(n) = T(n-2)+2$ becomes

$= (T(n-3)+1)+2$

$T(n) = T(n-3)+3$

$\vdots$

$T(n) = T(n-k)+k \quad \ldots (1)$

If $k = n$ then equation (1) becomes

$T(n) = T(0)+n$

$= 0+n \quad \because$ Initial condition $T(0) = 0$

$T(n) = n$

$\therefore$ We can denote T(n) in terms of big oh notation as **T(n) = O(n).**

Example 1.8.2 *Solve the recurrence relation*

$T(n) = 2T\left(\frac{n}{2}\right)+n$

Solution : With $T(1) = 1$ as initial condition

$T(n) = 2\left(2T\left(\frac{n}{4}\right)+\frac{n}{2}\right)+n$

$T(n) = 4T\left(\frac{n}{4}\right)+2n$

$T(n) = 4\left(2T\left(\frac{n}{8}\right)+\frac{n}{4}\right)+2n$

$= 8T\left(\frac{n}{8}\right)+3n$

$T(n) = 2^3T\left(\frac{n}{2^3}\right)+3n$

$\vdots$

$T(n) = 2^kT\left(\frac{n}{2^k}\right)+kn$

$T(n) = n\cdot T\left(\frac{n}{n}\right)+\log n\cdot n$ If we assume $2^k = n$

$= n\cdot T(1)+n\cdot\log n$

$T(n) = n+n\log(n)$ $\because T(1) = 1$

i.e. $T(n) \approx n\log n$

Hence in terms of big oh notation -

$T(n) = O(n\log n)$

Example 1.8.3 *Solve the following recurrence relation -*

$T(n) = T\left(\frac{n}{3}\right)+C \qquad T(1) = 1$

Solution : Let,

$T(n) = T\left(\frac{n}{3}\right)+C$

$= \left(T\left(\frac{n}{9}\right)+C\right)+C$

$= T\left(\frac{n}{9}\right)+2C$

$= \left[T\left(\frac{n}{27}\right)+C\right]+2C$

$= T\left(\frac{n}{27}\right)+3C$

$= T\left(\frac{n}{3^k}\right)+kC$

If we put $3^k = n$ then

$= T\left(\frac{n}{n}\right)+\log_3 n\cdot C$

$= T(1)+\log_3 n\cdot C$

$T(n) = C\cdot\log_3 n+1$ $\because T(1) = 1$

Example 1.8.4 *Solve the recurrence relation by iteration :*

$T(n) = T(n-1)+n^4.$

Solution : Let

$$T(n) = T(n-1) + n^4$$

By backward substitution method,

$$T(n) = [T(n-2)] + (n-1)^4] + n^4 \qquad \because T(n-1) = T(n-2) + (n-1)^4$$

$$= T(n-2) + (n-1)^4 + n^4$$

$$= [T(n-3) + (n-2)^4] + (n-1)^4 + n^4$$

$$= T(n-3) + (n-2)^4 + (n-1)^4 + n^4$$

$$= [T(n-4) + (n-3)^4] + (n-2)^4 + (n-1)^4 + n^4$$

$$= T(n-4) + (n-3)^4 + (n-2)^4 + (n-1)^4 + n^4$$

$$\vdots$$

$$= T(n-k) + (n-k+1)^4 + (n-k+2)^4 + (n-k+3)^4 + \ldots n^4$$

If $k = n$ then

$$= T(n-n) + (n-n+1)^4\ (n-n+2)^4 + (n-n+3)^4 + \cdots + n^4$$

$$= T(0) + 1^4 + 2^4 + 3^4 + \ldots + n^4$$

$$= T(0) + \sum_{i=1}^{n} n^4$$

$$= T(0) + n(n^4) \qquad \because \sum_{i=1}^{n} n^4 = n^4 \cdot \sum_{i=1}^{n} 1 = n^4 \cdot n$$

$$= 0 + n^5 \qquad \because \text{Assume } T(0) = 0 \text{ as initial condition}$$

$\therefore$ $T(n) \approx \Theta(n^5)$

2) Tree Method

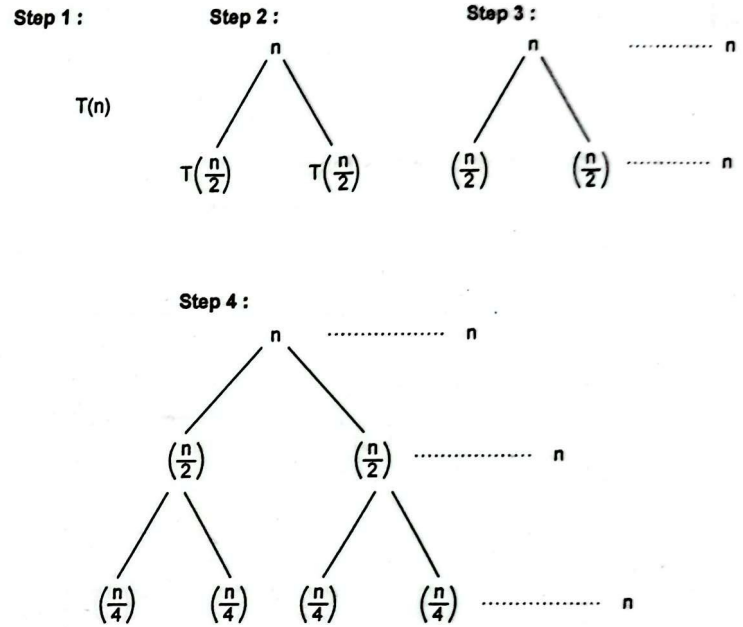
The recurrence relation can also be solved using tree method. In this method, a recursion tree is built in which each node represents the cost of a single subproblem in the form of recursive function invocations. Then we sum up the cost at each level to determine the overall cost. Thus recursion tree helps us to make a good guess of the time complexity. Let us understand this method with the help of some examples.

Example 1.8.5 *Solve the given recurrence relation using recursion tree.*

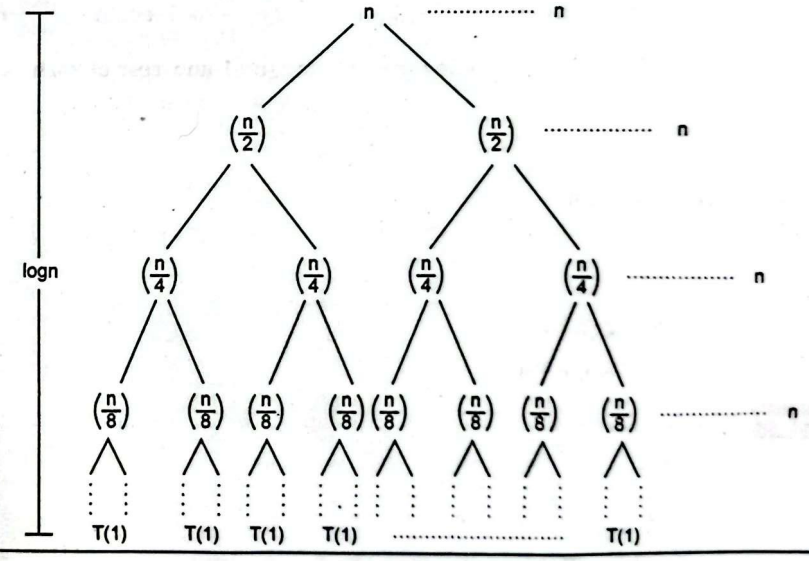
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(1) = O(1)$$

Solution :



After certain steps we will get



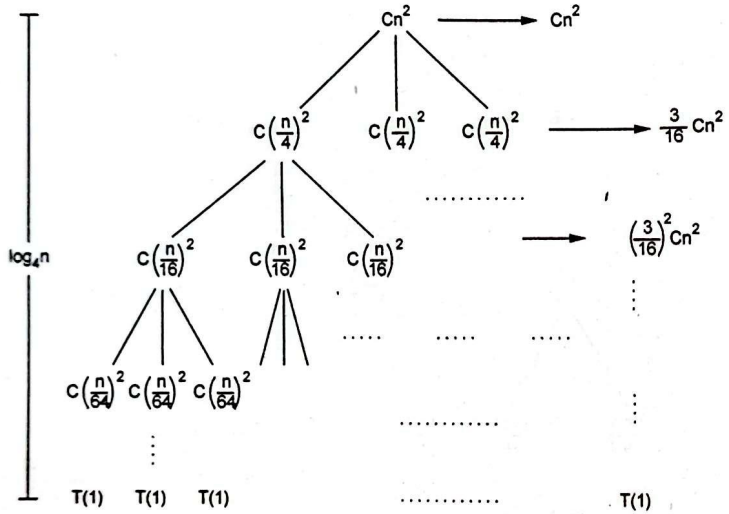
The depth of the tree is log n. Hence we can guess that,
Total cost = n log n T(1). Hence we get the overall cost as O(n log n).

Example 1.8.6 *Solve the recurrence relation*

$$T(n) = 3T\left(\frac{n}{4}\right) + Cn^2$$

using tree method.

Solution : The recursion tree can be drawn as follows :



Thus size for a node at depth i is $\frac{n}{4^i}$. If n = 1 the $\frac{n}{4^i} = 1$ i.e. $n = 4^i$. Hence $\log_4 n = i$. Thus depth i = (0, 1, 2, ... + $\log_4 n$+ 1) = $\log_4 n$+1 and cost of each node is $C\left(\frac{n}{4^i}\right)^2$.

$\therefore 3^i C\left(\frac{n}{4^i}\right)^2$ is overall cost

$$= \frac{3}{16} C(n^2)$$

The last level at dept $\log_4 n$ has

$$3^{\log_4 n} = n^{\log_4 3} = \theta(n^{\log_4 3})$$

Example 1.8.7 *Solve the recurrence -*

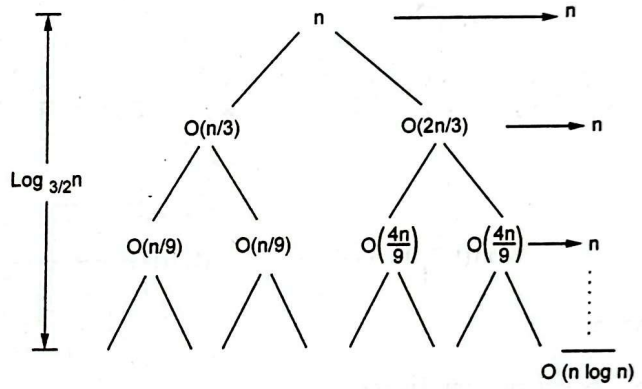
$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n) + \ldots$$

Solution : We will solve this recurrence using the tree method -

The longest path from root to a leaf is

$$n \rightarrow \left(\frac{2}{3}\right) n \rightarrow \left(\frac{2}{3}\right)^2 n \rightarrow \ldots \rightarrow 1$$

i.e. $\left(\frac{2}{3}\right)^k n = 1$



If we assume k = $\log_{3/2} n$. Then the height of the tree is $\log_{3/2} n$.

The solution to recurrence = Number of levels * cost of each level

$\therefore$ **T(n) = O(n log n)**

Example 1.8.8 *Find the closest asymptotic tight bound by solving the recurrence equation*

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2 \text{ with } T(1) = 1 \text{ using recursion tree method [Assume } T(1) \in \theta(1)]$$

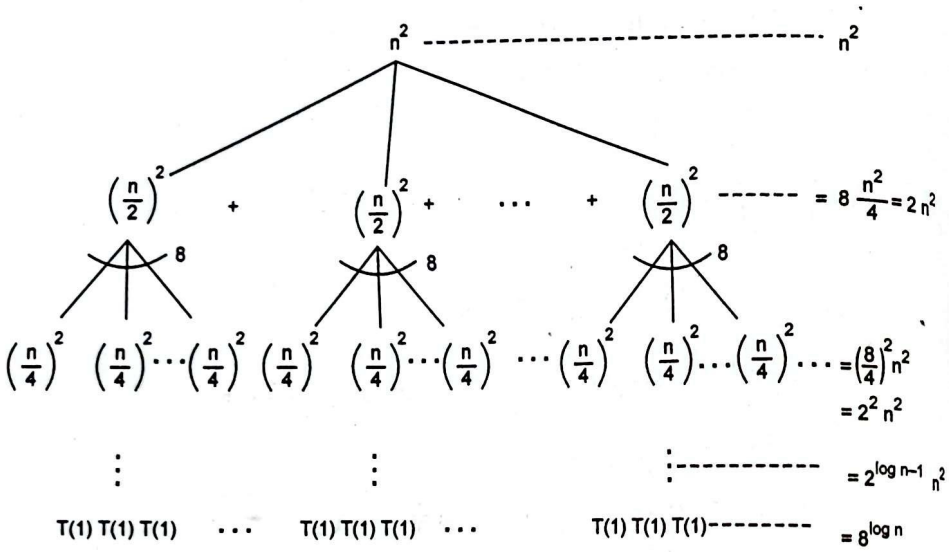
AU : Dec.-15, Marks 8

Solution : The recurrence relation can be written as

$$T(n) = n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + 2^4 n^2 + \ldots + 2^{\log n - 1} n^2 + 8^{\log n}$$

$$= \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n}$$

$$T(n) = n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}$$



From above equation

- $\sum_{k=0}^{\log n-1} 2^k$ is a geometric sum. Hence

$$\sum_{k=0}^{\log n-1} 2^k = \theta(2^{\log n-1}) = \theta(n)$$

$$(2^3)^{\log n} = (2^{\log n})^3 = n^3$$

$$\therefore \quad T(n) = n^2 \cdot \theta(n) + n^3$$

$$= \theta(n^3)$$

$$\therefore \quad \text{Time complexity} = \theta(n^3)$$

3) Master's Method

We can solve recurrence relation using a formula denoted by Master's method.

$$T(n) = aT(n/b) + F(n) \qquad \text{where } n \geq d \text{ and } d \text{ is some constant.}$$

Then the Master theorem can be stated for efficiency analysis as -

If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ in the recurrence relation then,

1. $T(n) = \Theta(n^d)$ if $a < b^d$
2. $T(n) = \Theta(n^d \log n)$ if $a = b$
3. $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

Let us understand the Master theorem with some examples :

Example 1.8.9 *Solve the following recurrence relation*

$T(n) = 4T(n/2) + n$

Solution : We will map this equation with

$$T(n) = aT(n/b) + f(n)$$

Now f(n) is n i.e. n^1. Hence d = 1.

$a = 4$ and $b = 2$ and

$a > b^d$ i.e. $4 > 2^1$

$$\therefore \quad T(n) = \Theta(n^{\log_b a})$$

$$= \Theta(n^{\log_2 4})$$

$$= \Theta(n^2) \qquad \because \log_2 4 = 2$$

Hence time complexity is $\Theta(n^2)$.

For quick and easy calculations of logarithmic values to base 2 following table can be memorized.

m	k
1	0
2	1
4	2
8	3
16	4
32	5

64	6
128	7
256	8
512	9
1024	10

$$\log_2 m = k$$

Another variation of Master theorem is

For $T(n) = aT(n/b) + f(n)$ if $n \geq d$

1. If $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then
   $T(n) = \Theta(n^{\log_b a})$
2. If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then
   $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then
   $T(n)$ is $= \Theta(f(n))$

Example 1.8.10 *Solve the following recurrences completely.*

1) $T(n) = \sum_{i=1}^{n-1} T(i) + 1$ *if* $n \geq 2$

$T(n) = 1$ *if* $n = 1$

2) $T(n) = 5T(n-2) - 6T(n-2)$

3) $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$

AU : Dec.-10, Marks 4+3+3

Solution : 1) Let,

$$T(n) = \sum_{i=1}^{n-1} T(i) + 1$$

$\therefore T(n) = T(n-1) + 1$

Putting value of equation (1)

$T(n) = (T(n-2) + 1) + 1$

By backward substitution,

$T(n-1) = T(n-2) + 1$ ... (1)

$\therefore T(n) = T(n-2) + 2$

$\because$ equation (2)

$T(n) = (T(n-3) + 1) + 2$

$$T(n) = T(n-2) + 2$$

Again $T(n-2) = T(n-3) + 1$ ... (2)

$\therefore$ $T(n) = T(n-3) + 3$

$\vdots$

$T(n) = T(n-k) + k$ ... (3)

If we put $n - k = 1$ then $k = n - 1$

Then equation (3) becomes

$T(n) = T(1) + n - 1$

$T(n) = 1 + n - 1$ $\because T(n) = 1$ if $n = 1$

$T(n) = n$

2) Let,

$T(n) = 5T(n-1) - 6T(n-2)$ be the given recurrence relation. We assume the initial condition as $a_0 = 0$ and $a_1 = 1$.

Step 1 : $T(n) = 5T(n-1) - 6T(n-2)$ can also be written in the form of characteristic equation as

$$a_n = 5a_{n-1} - 6a_{n-2}$$

i.e. $a_n - 5a_{n-1} + 6a_{n-2} = 0$ ... (4)

For $n \geq 2$ with initial condition $a_0 = 0$ and $a_1 = 1$ substitute $a_n = \lambda^n$.

Then equation (4) becomes,

$\lambda^2 - 5\lambda + 6 = 0$

i.e. $(\lambda - 2)(\lambda - 3) = 0$

$\therefore$ The solution for $\lambda = 2$ and 3.

Step 2 : Put $a_n = A(2)^n + B(3)^n$ ... (5)

Now we will find the values of A and B which satisfy the values of initial condition.

Put n = 0 in equation (5),

$$a_0 = A(2)^0 + B(3)^0$$

$$a_0 = A + B$$

$$\therefore \quad A + B = 0 \quad \text{... (6)}$$

Put n = 1 in equation (5),

$$a_1 = A(2)^1 + B(3)^1$$

$$a_1 = 2A + 3B$$

$$\therefore \quad 2A + 3B = 1 \quad \text{... (7)}$$

Solving equation (6) and (7) simultaneously we get,

$$A = -1$$

$$B = 1$$

Now equation (5) becomes,

$$\therefore \quad a_n = -2^n + 3^n \text{ is the solution.}$$

3) Let

Here $f(n) = n \log n$

$a = 2, \; b = 2$

$\log_2 2 = 1$

According to case 2 given in above Master theorem

$$f(n) = \Theta(n^{\log_2 2} \log^1 n) \qquad \text{i.e. } k = 1$$

Then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$$= \Theta(n^{\log_2 2} \log^2 n)$$

$$= \Theta(n^1 \log^2 n)$$

$$\therefore \quad T(n) = \Theta(n \log^2 n)$$

Example 1.8.11 *Solve the following recurrence relation*
$T(n) = 8T(n/2) + n^2$

Solution : Here $f(n) = n^2$

$a = 8$ and $b = 2$

$\therefore \quad \log_2 8 = 3$

Then according to case 1 of above given Master theorem

$$f(n) = O(n^{\log_b a - \varepsilon})$$

$$= O(n^{\log_2 8 - \varepsilon})$$

$$= O(n^{3-\varepsilon}).$$ If we put $\varepsilon = 1$ then $O(n^3 - 1) = O(n^2) = f(n)$

Then $T(n) = \Theta(n^{\log_b a})$

$\therefore \quad T(n) = \Theta(n^{\log_2 8})$

$$T(n) = \Theta(n^3)$$

Example 1.8.12 *Solve the following recurrence relation*
$T(n) = 9T(n/3) + n^3$

Solution : Here $a = 9$, $b = 3$ and $f(n) = n^3$

And $\log_3 9 = 2$

According to case 3 in above Master theorem

As $f(n)$ is $= \Omega(n^{\log_3 9 + \varepsilon})$

i.e. $\Omega(n^{2+\varepsilon})$ and we have $f(n) = n^3$.

Then to have $f(n) = \Omega(n)$. We must put $\varepsilon = 1$.

Then $T(n) = \Theta(f(n))$

$$T(n) = \Theta(n^3)$$

Example 1.8.13 *Solve the following recurrence relation.*
$T(n) = T(n/2) + 1$

Solution : Here $a = 1$ and $b = 2$.

and $\log_2 1 = 0$

Now we will analyze f(n) which is = 1.

We assume $f(n) = 2^k$

when $k = 0$ then $f(n) = 2^0 = 1$.

That means according to case 2 of above given Master theorem.

$$f(n) = \Theta(n^{\log_b a} \log^k n)$$

$$= \Theta(n^{\log_2 1} \log^0 n)$$

$$= \Theta(n^0 \cdot 1)$$

$$= \Theta(1) \qquad \because n^0 = 1$$

$\therefore$ We get $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$$= \Theta(n^{\log_2 1} \log^{0+1} n)$$

$$= \Theta(n^0 \cdot \log^1 n)$$

$$T(n) = \Theta(\log n) \qquad \because n^0 = 1$$

Example 1.8.14 *Find the complexity of the following recurrence relation.*
$T(n) = 9T(n/3) + n$

Solution : Let $T(n) = 9T(n/3) + n$

$\downarrow \quad \downarrow \quad \downarrow$

a b f(n)

$\therefore$ We get $a = 9$, $b = 3$ and $f(n) = n$.

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Now $f(n) = n$

i.e. $T(n) = f(n) = \Theta(n^{\log_b a - \varepsilon}) = \Theta(n^{2-\varepsilon})$

When $\varepsilon = 1$. That means case 1 is applicable. According to case 1, the time complexity of such equations is

$$T(n) = \Theta(n^{\log_b a})$$

$$= \Theta(n^{\log_3 9})$$

$$T(n) = \Theta(n^2)$$

Hence the complexity of given recurrence relation is $\Theta(n^2)$.

Example 1.8.15 *Solve recurrence relation*
$T(n) = k \cdot T(n/k) + n^2$ *when* $T(1) = 1$, *and* k *is any constant.*

Solution : Let,

$$T(n) = k \cdot T\left(\frac{n}{k}\right) + n^2 \text{ be a recurrence relation.}$$

If we assume $k = 2$ then the equation becomes

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2 \qquad T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2$$

$$= 2\left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n^2}{4}\right] + n^2$$

$$= 4T\left(\frac{n}{4}\right) + \frac{n^2}{2} + n^2$$

$$= 4T\left(\frac{n}{4}\right) + \frac{3n^2}{2} \qquad T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2$$

$$= 4\left[2 \cdot T\left(\frac{n}{8}\right) + \frac{n^2}{16}\right] + \frac{3n^2}{2}$$

$$= 8T\left(\frac{n}{8}\right) + \frac{n^2}{4} + \frac{3n^2}{2}$$

$$= 8T\left(\frac{n}{8}\right) + \frac{7n^2}{4}$$

$$= 2^k T\left(\frac{n}{2^k}\right) + \frac{2^k - 1}{2^{k-1}} n^2$$

$$= 2^k T\left(\frac{n}{2^k}\right) + Cn^2 \qquad \because \frac{2^k - 1}{2^{k-1}} = c = \text{Constant}$$

If we put $\frac{n}{2^k} = 1$ then $2^k = n$ and $k = \log_2 n$

$$= nT(1) + Cn^2$$

$$= n + Cn^2 \qquad \because T(1) = 1$$

$$T(n) = \Theta(n)^2$$

Alternate Method

We can solve the given recurrence relation using Master's theorem also.

$$T(n) = k \cdot T\left(\frac{n}{k}\right) + n^2$$

$$\downarrow \quad \downarrow \quad \downarrow$$

$$a \quad b \quad f(n)$$

By Master's theorem $n^{\log_b a} = n^{\log_k k} = n^1$

For $T(n) = f(n)$ we need

$T(n) = n^{\log_k k + \varepsilon}$ Applying case 3

$= n^{1+1}$ when $\varepsilon = 1$

$= f(n)$ i.e. n^2

$\therefore \quad T(n) = \Theta(f(n))$

$T(n) = \Theta(n^2)$

Example 1.8.16 *Solve the following recurrence relations :*

1) $T(n) = \begin{cases} 2T(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$

2) $T(n) = \begin{cases} 2T(n/2) + cn & n > 1 \\ \alpha & n = 1 \end{cases}$

where α and c are constants. **AU : Dec.-12, Marks (4+4)**

Solution :

1) Let, $T(n) = 2T\left(\frac{n}{2}\right) + 3$

By forward substitution method. We will solve above recurrence.

$T(2) = 2$

$T(4) = 2T\left(\frac{4}{2}\right) + 3 = 2T(2) + 3$

$T(4) = 2(2) + 3 = 7$

$T(8) = 2T\left(\frac{8}{2}\right) + 3$

$T(8) = 2(7) + 3 = 17$

$T(16) = 2T\left(\frac{16}{2}\right) + 3 = 2T(8) + 3$

$T(16) = 2(17) + 3 = 37$

$\vdots$

$\vdots$

$T(2n) = 2n + c$ where c is constant

Hence $T(n) \approx O(2n + c) \approx O(n)$

2) Let, $T(n) = 2T\left(\frac{n}{2}\right) + cn$

Assume $T(1) = \alpha = 0$

If we put $n = 2^k$ then

$$T(n) = 2T\left(\frac{2^k}{2}\right) + c \cdot 2^k$$

$$T(2^k) = 2T(2^{k-1}) + c \cdot 2^k$$

If we put $k = (k - 1)$ then

$$T(2^k) = 2(2\,T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k$$

$$= 2^2 T(2^{k-2}) + 2 \cdot c \cdot \frac{2^{k-1}}{2} + c \cdot 2^k$$

$$= 2^2 T(2^{k-2}) + 2 \cdot c \cdot 2^k$$

$\therefore \quad T(2^k) = 2^2\,T\left(2^{k-2}\right) + 2 \cdot c \cdot 2^k$

Similarly we can write

$$T(2^k) = 2^3\,T(2^{k-3}) + 3 \cdot c\,2^k$$

$$= 2^4 T\,(2^{k-4}) + 4 \cdot c\,2^k$$

...

...

...

$$= 2^k\,T(2^{k-k}) + k \cdot c\,2^k$$

$$= 2^k\,T\left(2^0\right) + k \cdot c \cdot 2^k$$

$T(2^k) = 2^k\,T(1) + k \cdot c\,2^k$

$T(2^k) = 2^k\,0 + k \cdot c \cdot 2^k$ $\because T(1) = 0$

$T(2^k) = k \cdot c \cdot 2^k$

But we assumed $n = 2^k$, hence $k = \log_2 n$

$\therefore$ $T(n) = \log_2 n \cdot c \cdot n$

$T(n) = \Theta (n \log_2 n)$

Example 1.8.17 *Suppose W satisfies the following recurrence equation and base case (Where c is a constant) : W(n) = c.n + W(n/2) and W(1) = 1. what is the asymptotic order of W(n).*

AU : Dec.-15, Marks 6

Solution : Let,

$$W(n) = W\left(\frac{n}{2}\right) + c.n \quad \ldots (1)$$

$$= \left(W\left(\frac{n}{4}\right) + c.n\right) + c.n$$

$$= \left(W\left(\frac{n}{8}\right) + c.n\right) + 2cn$$

$$= W\left(\frac{n}{2^3}\right) + 3cn$$

$$= W\left(\frac{n}{2^4}\right) + 4cn$$

$$\vdots$$

$$W(n) = W\left(\frac{n}{2^k}\right) + kcn \quad \ldots (2)$$

If we assume $2^k = n$ then equation 2 becomes

$$W(n) = W\left(\frac{n}{n}\right) + kcn$$

$$= W(1) + kcn$$

$$W(n) = 1 + kcn$$

$$W(n) = nc.\log n$$

$\therefore$ Time complexity **O (n log n)**

Example 1.8.18 *Solve the following recurrence relations.*

- $x(n) = x(n-1) + 5$ *for* $n>1$ $x(1) = 0$
- $x(n) = 3x(n-1)$ *for* $n>1$ $x(1) = 4$
- $x(n) = x(n-1) + n$ *for* $n>0$ $x(0) = 0$
- $x(n) = x(n/2) + n$ *for* $n>1$ $x(1) = 1$ *(Solve for* $n = 2^k$*)*
- $x(n) = x(n/3) + 1$ *for* $n>1$ $x(1) = 1$ *(Solve for* $n = 3^k$*)*

AU : Dec.-16, Marks 16

Solution : i) $x(n) = x(n-1) + 5$

By backward substitution,

$x(n) = (x(n-2) + 5) + 5$

$= x(n-2) + 2 * 5$

$= (x(n-3) + 5) + 2 * 5$

$= x(n-3) + 3 * 5$

.

.

.

$= x(n-k) + k * 5$

.

.

.

$= x(n-(n-1)) + (n-1) * 5$

$= x(1) + (n-1) * 5$

$= 0 + 5 * (n-1)$ $\because x(1) = 0$

$\therefore$ $\mathbf{x(n) = 5(n-1)}$

ii) $x(n) = 3x(n-1)$

$= 3(3x(n-2))$

$= 3^2 x(n-2)$

$= 3(3^2 x(n-3)$

$= 3^3 x(n-3)$

.

.

.

$= 3^k x(n-k)$

if $k = n-1$ then

$x(n) = 3^{n-1} x(n-(n-1))$

$= 3^{n-1}x(1)$

$= 3^{n-1} * 4$ $\because x(1) = 4$

$\therefore \quad x(n) = 4 * 3^{n-1}$

iii) $x(n) = x(n-1) + n$

$= (x(n-2) + n) + n$

i.e. $= x(n-2) + 2n$

$= (x(n-3) + n) + 2n$

i.e $= x(n-3) + 3n$

.

.

.

$= x(n-k) + kn$

if $k = n$ then

$x(n) = x(n-n) + n \cdot n$

$= x(0) + n^2$

$= 0 + n^2$ $\because x(0) = 0$

$\therefore \quad x(n) = n^2$

iv) $x(n) = x(n/2) + n$

Put $n = 2^k$ then

$x(2^k) = x(2^k / 2) + 2^k$

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

$= [x(2^{k-2}) + 2^{k-1}] + 2^k$

$= x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k$

=

.

.

.

$= x(2^{k-k}) + 2^1 + 2^2 + 2^3 + \ldots + 2^k$

$= x(2^0) + 2^1 + 2^2 + 2^3 + \ldots + 2^k$

$= x(1) + 2^1 + 2^2 + 2^3 + \ldots + 2^k$

$= 1 + 2^1 + 2^2 + 2^3 + \ldots 2^k$ $\because x(1) = 1$

$= 2^{k+1} - 1$

$= 2 \cdot 2^k - 1$

As $2^k = n$ then

$x(2^k) = 2 \cdot n - 1$

Let, $x(n) = x(n/3) + 1$

Assume $n = 3^k$

$x(3^k) = x(3^{k-1}) + 1$

$= [x(3^{k-2}) + 1] + 1$

$= x3^{k-2} + 2$

$= [x(3^{k-3}) + 1] + 2$

$= x \cdot 3^{k-3} + 3$

...

$= x(3^{k-k}) + k$

$= x(1) + k$

$= 1 + k$ $\because x(1) = 1$

But $3^k = n$ i.e. $k = \log_3 n$

$\therefore \quad x(3^k) = 1 + \log_3 n$

University Questions

1. *Define recurrence equation and explain how solving recurrence equations are done.*
   AU : Dec.-11, Marks 6
2. *With a suitable example, explain the method of solving recurrence equations.*
   AU : May-12, Dec.-14, Marks 16

1.9 Best Case, Worst Case and Average Case

- **Best case Time Complexity :** If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case time complexity. For example - While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.
- **Worst Case Time Complexity :** If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called worst case time complexity. For example - While searching an element by using linear searching method if desired element is placed at the end of the list then we get worst time complexity.
- **Average Case Time Complexity :** The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called average case time complexity.
- **Algorithmic Example :** Consider following algorithm. This algorithm is for sequential search.

```
Algorithm Seq_search(A[0 ... n-1 ],key)
// Problem Description: This algorithm is for searching the
//key element from an array A[0...n-1] sequentially.
for i=  0 to n-1 do
 if(A[i]=key)then
 return i
```

Analysis

1. **Best Case Analysis :** In above searching algorithm the element key is searched from the list of n elements. If the key element is present at **first location** in the list(A[0...n-1]) then algorithm **run for a very short time** and thereby we will get the best case time complexity. Hence in terms of big-oh notation the time complexity can be denoted as

   $$T(n) = O(1)$$

2. **Worst Case Analysis :** Worst case time complexity is a time complexity when algorithm runs for a longest time. In above searching algorithm the element key is searched from the list of n elements. If the key element is present at nth location then clearly the algorithm will run for longest time and thereby we will get the worst case time complexity. We can denote the worst case time complexity as

   $$T(n) = n$$

Hence in terms of big-oh notation the time complexity can be denoted as

$$T(n) = O(n)$$

3. **Average Case Analysis :** This type of complexity gives information about the behaviour of an algorithm on specific or random input. We can compute the average case time complexity as follows -

Let,

P be a probability of getting successful search.

n is the total number of elements in the list.

- The first match of the element will occur at ith location. Hence probability of occurring first match is P/n for every i^{th} element.
- The probability of getting unsuccessful search is (1 – P).
- Now, we can find average case time complexity T(n) as -

T(n) = Probability of successful search + Probability of unsuccessful search

For elements 1 to n in the list

When there is possibility of not getting element

$$T(n) = \left(1\frac{P}{n}+2\frac{P}{n}+.....+i\frac{P}{n}\right)+n\cdot(1-P)$$

$$= \frac{P}{n}(1+2+....+i)+n\cdot(1-P)$$

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

$$\boxed{T(n) = P\frac{(n+1)}{2}+n\cdot(1-P)} \quad \ldots (1)$$

This is general formula for computing average case time complexity.

In equation (1) we will put different values of P

If P = 0, we get

$$T(n) = \frac{0\,(n+1)}{2}+n\,(1-0)$$

$$\mathbf{T(n) = n}$$

If P=1, we get

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

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

That means the algorithm scans about the half of the elements from the list.

Hence time complexity can be denoted in terms of big oh notation as

$$T(n) = O(n)$$

Hence we can summarize the time complexity of sequential search in best case, worst case and average case in the form of Big-oh notation as

Best Case	Worst Case	Average Case
O(1)	O(n)	O(n)

1.10 Empirical Analysis

Definition : Empirical analysis of algorithm means observing behaviour of algorithm for certain set of input.

In empirical analysis actual program is written for the corresponding algorithm with the help of some suitable input set, the algorithm is analyzed.

General Plan for Empirical Analysis of Algorithm

1. Understand the purpose of experiment of given algorithm.
2. Decide the efficiency metric M. Also decide the measurement unit. For example operation's count Vs time.
3. Decide on characteristics of the input.
4. Create a program for implementing the algorithm. This program is ready for experiment.
5. Generate a sample of input.
6. Run the algorithm for some set of input sample. Record the results obtained.
7. Analyze the resultant data.

Let us discuss each step in detail –

1. Understand the purpose of experiment of given algorithm.

The algorithm is analyzed empirically for following reasons –

- The accuracy of an algorithm needs to be checked.
- For the same problem there can be several algorithms available and one can compare the efficiencies of different algorithms for same problem. Or one can compare different implementations for the same problem.

- One can decide the efficiency of particular algorithm on particular machine.
- The hypothesis for algorithm's efficiency class can be developed

2. Decide the efficiency metric M. Also decide the measurement unit.

The efficiency of an algorithm can be measured by following different methods –

- Insert a counter in the algorithm in order to count the number of times the basic operation is executed. This is a straightforward method of counting an efficiency of algorithm.

For example : If we write a function for calculating sum of n number in an array then we can find the efficiency of that function by inserting a **frequency count.** The frequency count is a count that denotes how many times the particular statement is executed.

```
Line 1: int sum element(int a[10],int n)
Line 2: {
Line 3:   int i, sum=0;
Line 4:   for(i=0;i<n;i++)
Line 5:   {
Line 6:     sum=sum+a[i];
Line 7:   }
Line 8:   return sum;
Line 9: }
```

The execution starts from **for** loop. The declaration part can be neglected. Now

Statement	Frequency count
i=0	1
i<n	This statement executes for (n+1) times. When condition is true i.e. when i<n is true, the execution happens to be n times and the statement executes once more when i<n is false.
i++	n times
sum=sum+ a[i]	n times
return sum	1
Total	**(3n+3)**

We get the frequency count to be (3n+3). We can neglect the constant terms and in terms of asymptotic notations the efficiency of an algorithm can be denoted as **O(n)**.

Now the program for above algorithm can be written in suitable programming language and run it for various input cases.

3. Decide on characteristics of the input.

Even though the efficiency of basic operation is measured by frequency count or by time clocking it is necessary to consider some set of inputs for experiment. For certain

sample size of input the behaviour of an algorithm is to be observed. For example it double the sample size then we can compute the ratios M(2n)/M(n) of observed me M. The metric can be time or a frequency count. From this observed ratio we identify the basic efficiency class of that algorithm. Sometimes based on random in the empirical analysis of algorithm is done. Thus range or size of the input is decided analyze the algorithm.

4. Generate a sample of input.

For the decided range of input the various samples are obtained.

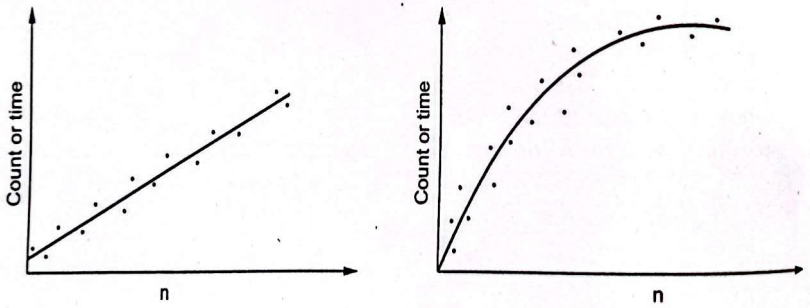
5. Run the algorithm for some set of input sample. Record the results obtained.

Using some suitable programming language the program is to be written for gi algorithm and for some set of input sample the program is run. The results obtained certain set of input is to be recorded. This recorded result is then presented for analys

6. Analyze the resultant data.

While analyzing the resultant data first arrange it in tabular or in graphical form.

For example :



(a) Input obtained gives linear characteristic The efficiency = M(n)

(b) Input obtained gives logarithmic graph. The efficiency = $M(\log_2 n)$

Fig. 1.10.1

Example 1.10.1 *Using Step count method analyze the time complexity when two m matrices are added.*

```
void fun(int a[][],int b[][])
{
 int c[3][3];
  for(i=0;i<m;i++)
  {
    for(j=0;j<n;j++)
    {
      c[i][j]=a[i][j]+b[i][j];
    }
}
```


Solution :

```
void fun(int a[][],int b[][])
{
int c[3][3];
 for(i=0;i<m;i++) .....m+1
 {
  for(j=0;j<n;j++) .....m(n+1)
  {
   c[i][j]=a[i][j]+b[i][j]; .....m.n
  }
}
```

The frequency count = (m + 1) + m(n + 1) + mn = 2m + 2mn + 1 = 2m(1 + n) + 1

Example 1.10.2 *Obtain the frequency count for the following code. for(i=1;i<=n;i++)*

```
 {
 for(j=1;j<=n;j++)
 {
 c[i][j]=0;
 for(k=1;k<=n;k++)
 c[i][j]=c[i][j]+a[i][k]*b[k][j];
 }
 }
```

Solution :

Statement	Frequency Count
for(i=1;i<=n;i++)	n + 1
for(j=1;j<=n;j++)	n.(n + 1)
c[i][j]=0;	n.(n)
for(k=1;k<=n;k++)	n.n(n + 1)
c[i][j]=c[i][j]+a[i][k]*b[k][j];	n.n.n
Total	$2n^3 + 3n^2 + 2n + 1$

After counting the frequency count, the constant terms can be neglected and only t order of magnitude is considered. The time complexity is denoted in terms algorithmic notations. The Big oh notation is a most commonly used algorithm notation. For the above frequency count all the constant terms are neglected and on the order of magnitude of the polynomial is considered. Hence the time complexity f the above code can be $O(n^3)$. The higher order is the polynomial is always considered.

Difference between Mathematical and Empirical Analysis

Sr. No.	Mathematical Analysis	Empirical Analysis
1.	The algorithm is analyzed with the help of *mathematical derivations* and there is no need of specific input.	The algorithm is analyzed by taking some *sample of input* and no mathematical derivation is involved.
2.	The principal weakness of this type of analysis is its *limited applicability*.	The principal strength of empirical analysis is it is *applicable* to any algorithm.
3.	The principal strength of mathematical analysis is it is *independent* of any input or the computer on which the algorithm is running.	The principal weakness of empirical analysis is that it *depends* upon the sample input taken and the computer on which the algorithm is running.

1.11 Mathematical Analysis for Recursive Algorithms

AU : May-11,14,16,18 Dec.-12,13,17, Marks 16

1.11.1 General Plan

1. Decide the **input size** based on parameter n.
2. Identify algorithm's basic operation(s).
3. Check how many times the basic operation is executed. Then find whether th **execution of basic operation** depends upon the input size n. Determine worst **average, and best cases** for input of size n. If the basic operation depends upo worst case, average case and best case then that has to be analyzed separately.
4. Set up the **recurrence relation** with some initial condition and expressing the basi operation.
5. Solve the recurrence or atleast determine the order of growth. While solving th recurrence we will use the **forward and backward substitution** method. And the correctness of formula can be proved with the help of **mathematical inductio** method.

Let us analyze some recursive algorithms mathematically.

1.11.2 Examples

1. Computing factorial of some number n.

The factorial of some number can be obtained by performing repeated multiplication.

For instance : If n = 5 then

Step 1 : n! = 5!

Step 2 : 4! * 5

Step 3 : 3! * 4 * 5

Step 4 : 2! * 3 * 4 * 5

Step 5 : 1! * 2 * 3 * 4 * 5

Step 6 : 0! * 1 * 2 * 3 * 4 * 5

Step 7 : 1 * 1 * 2 * 3 * 4 * 5 as 0! = 1

The above mentioned steps can be written in pseudocode form as -

```
Algorithm Factorial (n)
//Problem Description : This algorithm computes nl using
//recursive function
//Input : A non negative integer n

//Output : returns the factorial value.
        if (n=0)
                return 1
        else
                return Factorial (n-1)*n
```

Mathematical Analysis

Step 1 : The factorial algorithm works for input size n.

Step 2 : The basic operation in computing factorial is multiplication.

Step 3 : The recursive function call can be formulated as

$F(n) = F(n-1) * n$ where $n > 0$

Then the basic operation multiplications is given as M(n). And M(n) is multiplication count to compute factorial (n).

$$M(n) = M(n-1) + 1$$

These multiplications are required to compute factorial (n – 1)

To multiply factorial (n – 1) by n.

Step 4 : In step 3 the recurrence relation is obtained.

$$M(n) = M(n-1) + 1$$

Now we will solve recurrence using

- **Forward substitution**

$$M(1) = M(0) + 1$$

$$M(2) = M(1) + 1 = 1 + 1 = 2$$

$$M(3) = M(2) + 1 = 2 + 1 = 3$$

- **Backward substitution**

$$M(n) = \underbrace{M(n-1)}_{\downarrow} + 1$$

$$= \left[\underbrace{M(n-2)}_{\downarrow} + 1\right] + 1 = M(n-2) + 2$$

$$= [M(n-3) + 1] + 1 + 1 = M(n-3) + 3$$

From the substitution methods we can establish a general formula as :

$$M(n) = M(n-i) + i$$

Thus the time complexity of factorial function is $\Theta(n)$.

2. *Towers of Hanoi*

The problem "Towers of Hanoi" is a classic example of recursive function. The problem can be understood by following discussion. The initial setup is as shown in Fig. 1.11.1.

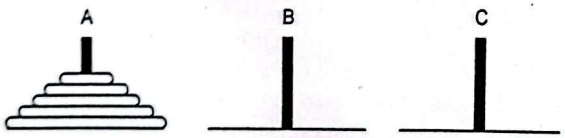


Fig. 1.11.1

There are three pegs named as A, B and C. The five disks of different diameters are placed on peg A. The arrangement of the disks is such that every smaller disk is placed on the larger disk.

The problem of " Towers of Hanoi" states that move the five disks from peg A to peg C using peg B as an auxiliary.

The conditions are :

i) Only the top disk on any peg may be moved to any other peg.

ii) A larger disk should never rest on the smaller one.

The above problem is the classic example of recursion. The solution to this problem is very simple.

First of all let us number out the disks for our comfort.

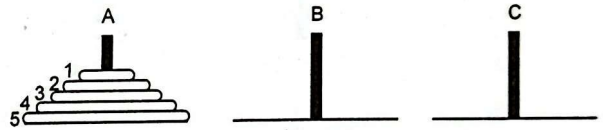


Fig. 1.11.2

The solution can be stated as

1. Move top n – 1 disks from A to B using C as auxiliary.
2. Move the remaining disk from A to C.
3. Move the n – 1 disks from B to C using A as auxiliary.

We can convert it to

move disk 1 from A to B.
move disk 2 from A to C.
move disk 1 from B to C.

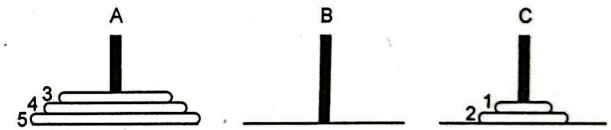


Fig. 1.11.3

move disk 3 from A to B
move disk 1 from C to A
move disk 2 from C to B
move disk 1 from A to B

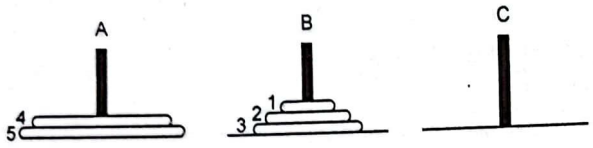


Fig. 1.11.4

move disk 4 from A to C
move disk 1 from B to C
move disk 2 from B to A
move disk 1 from C to A

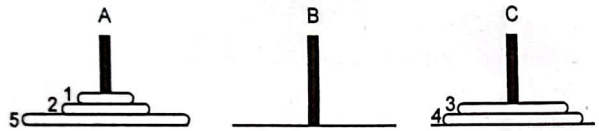


Fig. 1.11.5

move disk 3 from B to C
move disk 1 from A to B
move disk 2 from A to C

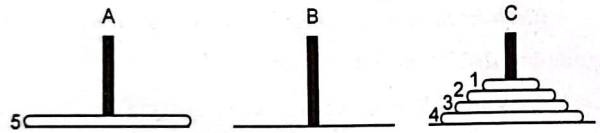


Fig. 1.11.6

move disk 1 from B to C

Thus actually we have moved n – 1 disks from peg A to C.

```
Algorithm TOH(n, A, C, B)
{
//if only one disk has to be moved
 if(n=1)then
 {
      write("The peg moved from A to C")
      return
 }
 else
 {
  //move top n-1 disks from A to B using C
  TOH(n-1,A,B,C);
  //move remaining disk from B to C using A
  TOH(n-1,B,C,A);
 }
}
```


Mathematical Analysis :

Step 1 : The input size is n i.e. total number of disks.

Step 2 : The basic operation in this problem is moving disks from one peg to another. When n > 1, then to move these disks from peg A to peg C using peg B, we first move recursively n – 1 disks from peg A to peg B using auxiliary peg C. Then we move the largest disk directly from peg A to peg C and finally move n – 1 disks from peg B to peg C (using peg A as auxiliary peg).

If n = 1 then we simply move the disk from peg A to peg C.

Step 3 : The moves of disks are denoted by M(n). M(n) depends on number of disks n. The recurrence relation can then set up as

$M(1) = 1$ ∵ Only 1 move is needed TOH (1, *, *, *,)

If n > 1 then we need two recursive calls plus one move. Hence

$$M(n) = M(n-1) + 1 + M(n-1)$$

To move (n–1)disks from peg A to B

To move largest disk from peg A to C

To move (n – 1) disks from peg B to C

$$\therefore \quad M(n) = 2M(n-1) + 1 \qquad \ldots (1)$$

Step 4 :

Solving recurrence M(n) = 2M(n – 1) + 1 using two substitution methods.

- **Forward substitution :**

For n > 1

$M(2) = 2M(1) + 1$
$= 2 + 1$
$M(2) = 3$
$M(3) = 2\,M(2) + 1$
$= 2\,(3) + 1$
$M(3) = 7$
$M(4) = 2\,M(3) + 1$
$= 2\,(7) + 1$
$M(4) = 15$

- **Backward substitution :**

$M(n) = 2\,M(n-1) + 1$

$= 2\,[2M(n-2) + 1] + 1$

$= 4M(n-2) + 3$

$= 4\,[2M(n-3) + 1] + 3$

$= 8\,M(n-3) + 7$

Put $M(n-1) = 2M(n-2) + 1$

This is also written as $2^2\,M(n-2) + 2 + 1$

Put $M(n-2) = 2M(n-3) + 1$

Can be written as $2^3\,M(n-3) + 2^2 + 2^1 + 1$

Above computations suggest us to compute next computation as

$$= 2^4\,M(n-4) + 2^3 + 2^2 + 2 + 1$$

From this we can establish a general formula as

$$M(n) = 2^i\,M(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2 + 1$$

This can also be written as

$$M(n) = 2^i\,M(n-i) + 2^i - 1 \quad \ldots (2)$$

Thus for obtaining M(n) we substitute n by n – i in the equation (2).

From this we can conclude that tower of Hanoi has a time complexity as $\Theta(2^n - 1) = \Theta(2^n)$.

The tree can be drawn for showing the recursive calls made in the problem of Tower of Hanoi.

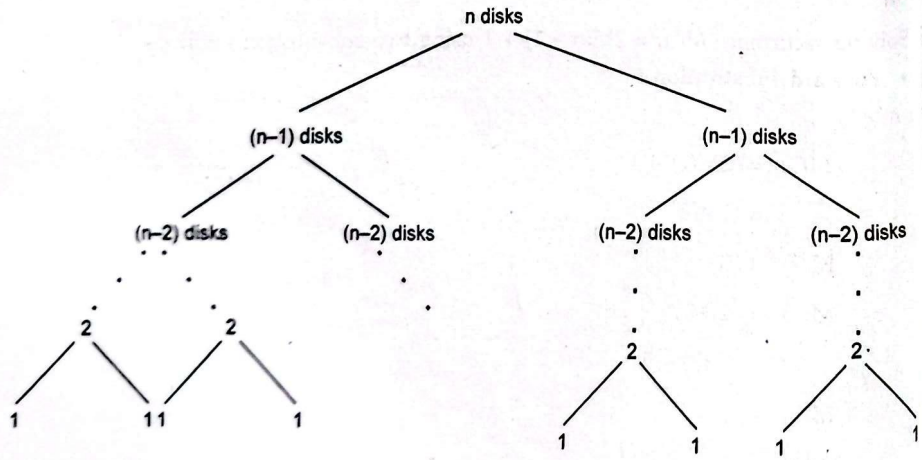


Fig. 1.11.7 Tree of recursive calls

3. *Number of bits in integer.*

In this algorithm we will count the number of binary digits (either 0 to 1) from the given decimal number.

```
Algorithm Binary_Rec (n)
//Problem Description: This algorithm is for counting the
//binary digits from a decimal integer
//Input: The decimal integer n
//Output: Returns total number of digits from the input
if(n=1) then
  return 1
else
  return(Binary_Rec (⌊n/2⌋) + 1
```

Mathematical Analysis

Step 1 : The input size = n.

Step 2 : The basic operation which is performed is division by 2.

Step 3 : Now we will set up recurrence relation for this algorithm.

Let D(n) be a count of performing division to calculate Binary_Rec(n).

When n = 1 then there is no division operation performed.

$\therefore\ D(1) = 0$

If n > 1 then Binary_Rec(n) makes a recursive call with $\lfloor n/2 \rfloor$

$D(n) = D(\lfloor n/2 \rfloor) + 1$

To compute Binary_Rec $\lfloor n/2 \rfloor$ — To compute $\lfloor n/2 \rfloor$

$\therefore\ D(n) = D(\lfloor n/2 \rfloor) + 1$ for n > 1

Step 4 : Solving recurrence for n to be powers of 2, we can compute the efficiency of an algorithm.

- **Forward substitution**

$D(2) = D(1) + 1$

$D(2) = 1 \qquad \because D(1) = 0$

Similarly $D(4) = D(2) + 1$

$= 1 + 1 \quad \because D(2) = 1$

$\therefore \quad D(4) = 2$

Similarly $D(8) = D(4) + 1$

$= 2 + 1 \quad \because D(4) = 2$

$D(8) = 3$

- **Backward substitution**

$D(n) = D(\lfloor n/2 \rfloor) + 1$ Substitute $D(\lfloor n/2 \rfloor) = D(\lfloor n/4 \rfloor) + 1$

$= \left[D(\underbrace{\lfloor n/4 \rfloor}_{\downarrow}) + 1 \right] + 1 = D(\lfloor n/4 \rfloor) + 2$

$= [D(\lfloor n/8 \rfloor) + 1] + 2 \quad \because D(\lfloor n/4 \rfloor) = D(\lfloor n/8 \rfloor) + 1$

$= D(\lfloor n/8 \rfloor) + 3$ This can be continued.

The standard approach is to solve the recurrence.

For $n = 2^k$ (i.e. in power of 2). Then using backward substitution we get,

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

To prove this using mathematical induction, we have to prove $D(2^k) = k$.

Basis : If $k = 0$ then

$D(2^k) = D(2^0)$

$= D(1)$

But as $D(1) = 0$ we get

$D(2^k) = k$ → We have assumed k to be 0 and we get $D(2^k) = 0$.

$\therefore \quad D(2^k) = k$ is proved.

Induction : Now assume $k = k - 1$ then

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

$= [D(2^{k-2}) + 1] + 1$ Substitute $D(2^{k-2}) = D(2^{k-3}) + 1$

$= [D(2^{k-3}) + 1] + 2 = [D(2^{k-3}) + 3]$

⋮

$= D(2^{k-i}) + i$

$= D(2^{k-k}) + k$

$= D(2^0) + k$

$= D(1) + k$ But $D(1) = 0$

$\therefore \quad D(2^k) = k$ is proved.

We have assumed $n = 2^k$. By taking log from both sides this can also be written as $k = \log_2 n$.

Thus we conclude that for counting number of bits from integer the time complexity required is $D(k) = \log_2 n \in \Theta(\log_2 n)$.

Example 1.11.1 *Write the recursive and non-recursive versions of the factorial function. Examine how much time each function requires as 'n' becomes large.*

AU : May-11, Marks 16

Solution :

```
/******** Recursive Program to find factorial of n number **********/
#include<stdio.h>
#include<conio.h>
void main()
{
 int n;
 printf("\n Enter the value of n");
 scanf("%d",&n);
 printf("\n The factorial is =%d",fact(n));
}
int fact(int n)
{
 int a,b;
 if(n==0)
  return 1;
 a=n-1;
 b=fact(a);
 return(n*b);
}
/******** Non-Recursive Program to find factorial of n number **********/
#include<stdio.h>
#include<conio.h>
void main()
{
 int n;
 printf("\n Enter the value of n");
```

```
 scanf("%d",&n);
 printf("\n The factorial is =%d",fact(n));
}
int fact(int n)
{
 int prod=1;
 int x;
 x=n;
 while(x>0)
 {
  prod=prod*x;
  x- -;
 }
 return prod;
}
```

In the recursive routine when each call is made, it gets stored in the stack the larger value of n may cause the stack overflow error.

Example 1.11.2 *Explain the recursive algorithm for computing Fibonacci numbers and analyze the same.* **AU : May-11, Marks 16**

Solution :

```
Algorithm Fib(n)
//Problem Description : The algorithm finds the Fibonacci number
//Input : The  number n for which the value of Fibonacci series is to be obtained
//Output : The Fibonacci number is returned
if (n<= 1) then
        return n,
else
return (Fib(n – 1) + Fib(n – 2))
```

Now to obtain formula for n^{th} Fibonacci number,

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

i.e. $F(n) - F(n-1) - f(n-2) = 0$

The characteristic equation for above equation is,

$$r^2 - r - 1 = 0$$

We can solve the quadratic equation by finding its roots.

The formula for finding roots of quadratic equation is $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

In,

$$r^2 - r - 1 = 0$$

$a = 1, b = -1, c = -1$, then the roots r_1 and r_2 are

$$r_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} = \frac{1+\sqrt{1-4(1*-1)}}{2(1)}$$

$$= \frac{1+\sqrt{1+4}}{2}$$

$$r_1 = \frac{1+\sqrt{5}}{2}$$

$$r_2 = \frac{-b-\sqrt{b^2-4ac}}{2a} = \frac{1-\sqrt{1-4(1*-1)}}{2}$$

$$= \frac{1-\sqrt{1+4}}{2}$$

$$r_2 = \frac{1-\sqrt{5}}{2}$$

Thus characteristic equation has two distinct roots. Then we will use following formula.

$$x(n) = Ar_1^n + Br_2^n \quad \text{... (1)}$$

where A and B are two constants.

We will put values of r_1 and r_2 in equation (1), then we obtain

$$F(n) = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n \quad \text{... (2)}$$

Now to obtain values of A and B we will make use of initial conditions.

As we know, initial conditions are

$$F(0) = 0$$

$$F(1) = 1$$

Then when n = 0, equation (2) becomes,

$$A\left(\frac{1+\sqrt{5}}{2}\right)^0 + B\left(\frac{1-\sqrt{5}}{2}\right)^0 = 0$$

i.e. $A + B = 0$ ... (3)

When n = 1, equation (2) becomes,

$$A\left(\frac{1+\sqrt{5}}{2}\right)^1 + B\left(\frac{1-\sqrt{5}}{2}\right)^1 = 1$$

i.e. $$A\left(\frac{1+\sqrt{5}}{2}\right) + B\left(\frac{1-\sqrt{5}}{2}\right) = 1 \quad \ldots (4)$$

From equation (3), we can write,

$$A = -B \quad \ldots (5)$$

Putting – B instead of A in equation (4) we get

$$-B\left(\frac{1+\sqrt{5}}{2}\right) + B\left(\frac{1-\sqrt{5}}{2}\right) = 1$$

i.e. $$B\left[\left(\frac{1-\sqrt{5}}{2}\right) - \left(\frac{1+\sqrt{5}}{2}\right)\right] = 1$$

i.e. $$B\left[\frac{1-\sqrt{5}-1-\sqrt{5}}{2}\right] = 1$$

i.e. $$B\left[\frac{-2\sqrt{5}}{2}\right] = 1$$

i.e. $$B[-\sqrt{5}] = 1$$

∴ We get $$B = -\frac{1}{\sqrt{5}}$$

$$A = -B$$

$$A = -\left(-\frac{1}{\sqrt{5}}\right)$$

$$A = \frac{1}{\sqrt{5}}$$

To put the values of A and B in equation (2), we will rewrite equation (2),

$$F(n) = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

If we assume $\phi = \frac{(1+\sqrt{5})}{2}$, $\phi^{-1} = \frac{(1-\sqrt{5})}{2}$ then

$$F(n) = \frac{1}{\sqrt{5}}(\phi)^n - \frac{1}{\sqrt{5}}(\phi^{-1})^n \quad \ldots (6)$$

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

$$\phi^{-1} = \frac{1-\sqrt{5}}{2} \approx -0.61803$$

The constant ϕ is called **golden ratio.**

The equation (6) can be finally,

$$F(n) = \frac{1}{\sqrt{5}}[(\phi)^n - (\phi^{-1})^n] \quad \ldots (7)$$

The term $\frac{1}{\sqrt{5}}(\phi^{-1})^n$ is infinitely small for infinite value of n. Hence we can take F(n) as,

$$F(n) = \frac{1}{\sqrt{5}}\phi^n$$

Thus $F(n) \in \Theta(\phi^n)$.

Example 1.11.3 *Find the Time complexity and space complexity of the following problems. Factorial using recursion AND compute n^{th} Fibonacci number using iterative statements.*

AU : Dec-12, Marks 8

Solution : i) Recursive Factorial Function

For Time Complexity - Refer section 1.11.1(1)

Space complexity : **O(n)** as internal stack is used to store n elements.

ii) Iterative Fibonacci Number : The iterative algorithm for Fibonacci function is

```
int fib(int n)
{
  f[0] = 0;
  f[1] = 1;
  for (i = 2; i <= n; i++)
  {
    /* Add the previous two numbers in the series and store it at i^th location*/
    f[i] = f[i-1] + f[i-2];
  }
 return f[n];
}
```

The basic operation in above code is performing addition in the for loop. This loop executes for n times. Hence the time complexity of this algorithm is **O(n)**.

The array of n elements is used to store the Fibonacci series. Hence the space complexity is **O(n)**.

University Questions

1. *Explain the Towers of Hanoi problem and solve it using recursion.* **AU : Dec.-13, May-14**
2. *Derive the recurrence relation for Fibonacci series algorithm; also carry out the time complexity analysis.* **AU : May-14**
3. *Give the recursive algorithm for finding the number of binary digits in n's binary representation, where n is a positive decimal integer. Find the recurrence relation and complexity.* **AU : May-16**
4. *Discuss the steps in Mathematical analysis for recursive algorithms. Do the same for finding the factorial of a number.* **AU : Dec.-17, Marks 13**
5. *Give the general plan for analyzing the time efficiency of recursive algorithms and use recurrence to find number of moves for towers of Hanoi problem.* **AU : May-18, Marks 13**

1.12 Mathematical Analysis for Non Recursive Algorithms

AU : Dec.-15,16,18, May-16,17, Marks 13

1.12.1 General Plan

1. Decide the **input size** based on parameter n.
2. Identify algorithm's **basic operation**(s).
3. Check how many times the basic operation is executed. Then find whether the **execution of basic operation** depends upon the input size n. Determine **worst, average and best cases** for input of size n. If the basic operation depends upon worst case, average case and best case then that has to be analyzed separately.
4. Set up a **sum** for the number of times the basic operation is executed.
5. **Simplify** the sum using standard formula and rules.

As mentioned in the step 5 we have to simplify the sum value using some standard formula and rules.

1.12.2 Examples

Now we will discuss a few non recursive algorithms. We will find the efficiency of these algorithms with the help of general plan.

1. Finding the element with maximum value in a given array.

```
Algorithm Max_Element(A[0...n–1])
//Problem Description: This algorithm is for finding the
//maximum value element from the array
//Input: array A[0...n–1]
//Output: Returns the largest element from array
 Max_value ← A[0]
for i ← 1 to n–1 do
 {
 if(A[i]>Max_value)then
          Max_value ← A[i]
 }
return Max_value
```

Searching the maximum element from an array

If any value is larger than current **Max_value** then set new **Max_value** by obtained larger value

Mathematical Analysis

Step 1 : The input size is n i.e. total number of elements in array.

Step 2 : The basic operation is comparison in loop for finding larger value.

Step 3 : The comparison is executed on each repetition of the loop. As the comparison is made for each value of n there is no need to find best case, worst case and average case analysis.

Step 4 : Let C(n) be the number of times the comparison is executed. The algorithm makes comparison each time the loop executes. That means with each new value of i the comparison is made. Hence for i = 1 to n – 1 times the comparison is made. Therefore we can formulate C(n) as -

C(n)= One comparison made for each value of i

Step 5 : Let us simplify the sum

$$C(n) = \sum_{i=1}^{n-1} 1$$

$$C(n) = n - 1 \in \Theta(n) \qquad \left[\text{Using the rule } \sum_{i=1}^{n} 1 = n \in \Theta(n)\right]$$

Thus the efficiency of above algorithm is Θ (n).

2. Finding whether all the elements in an array are distinct. This problem is called element uniqueness problem.

```
Algorithm Unique_Element(A[0...n–1])
//Problem Description: This algorithm finds whether array
//elements are distinct or not
//Input: Array A[0...n–1]
//Output: Returns False if elements are not distinct. If
//all the elements of an array are distinct then it returns
```

```
//True
for i ← 0 to n – 2 do
{
      for j ← i+1 to n–1 do
{
     if(A[i]=A[j]) then
          return False
}
}
return True
```

If any two elements in the array are similar then return False indicating that the array elements are not distinct.

Mathematical Analysis

Step 1 : The input size is n i.e. total number of elements in array A.

Step 2 : The basic operation will be comparison of two elements. This operation is the innermost operation in the loop. Hence

```
if (A[i]=A[j]) then
```

will be the basic operation.

Step 3 : The number of comparisons made will depend upon the input n. But the algorithm will have worst case complexity if the same element is located at the end of the list. Hence the basic operation depends upon the input n and worst case.

Step 4 : The worst case input is when we require largest number of comparisons for the array of size n. The worst case time is denoted by $C_{worst}(n)$. But there are two types of worst case inputs.

i) When there are no equal elements in the array.

ii) The last two elements are equal in the array.

For such type of inputs one comparison is made for each value of j (inner loop) ranging from i+1 to n – 1. This inner loop will be repeated for each value of i (outer loop) and i varies from 0 to n – 2. Hence we can obtain $C_{worst}(n)$ as -

$$C_{worst}(n) = \text{Outer loop} \times \text{Inner loop}$$

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Step 5 : *Now we will simplify* C_{worst} as follows -

$$\sum_{j=i+1}^{n-1} 1 = (n-1)-(i+1)+1 \qquad \left[\Theta \sum_{i=k}^{n} 1 = n-k+1 \text{ is the rule.}\right]$$

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

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

Now taking (n–1) as a common factor, we can write

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

This can be obtained using formula $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$$= (n-1)\left(\sum_{i=0}^{n-2} 1\right) - \frac{(n-2)(n-1)}{2}$$

This can be obtained using formula $\sum_{i=1}^{n} 1 = (n-1+1) = n$ i.e. when $\sum_{i=0}^{n-2} 1 = (n-2)-0+1 = (n-1)$

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

Solving this equation we will get

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

$$= \left(2(n^2-2n+1) - (n^2-3n+2)\right)/2$$

$$= (n^2-n)/2$$

$$\approx \frac{1}{2}n^2$$

$$\in \Theta(n^2)$$

We can say that in the worst case the algorithm needs to compare all n(n – 1)/2 distinct pairs of its n elements.

3. Obtaining matrix multiplication.

We have already learnt the method of matrix multiplication. While performing matrix multiplication we scan row of first matrix and column of second matrix.

For example :

$$C = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 1 & 2 & 3 \\ a_{10} & a_{11} & a_{12} \\ 4 & 5 & 6 \end{bmatrix}_{2\times 3} \times \begin{bmatrix} b_{00} & b_{01} \\ 1 & 2 \\ b_{10} & b_{11} \\ 3 & 4 \\ b_{20} & b_{21} \\ 5 & 6 \end{bmatrix}_{3\times 2}$$

The formula for multiplication of the above two matrices is

$$C = \begin{bmatrix} a_{00} \times b_{00} + a_{01} \times b_{10} + a_{02} \times b_{20} & a_{00} \times b_{01} + a_{01} \times b_{11} + a_{02} \times b_{21} \\ a_{10} \times b_{00} + a_{11} \times b_{10} + a_{12} \times b_{20} & a_{10} \times b_{01} + a_{11} \times b_{11} + a_{12} \times b_{21} \end{bmatrix}$$

$$C = \begin{bmatrix} 1\times1+2\times3+3\times5 & 1\times2+2\times4+3\times6 \\ 4\times1+5\times3+6\times5 & 4\times2+5\times4+6\times6 \end{bmatrix}$$

$$C = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}$$

Now the algorithm for matrix multiplication is -

```
Algorithm Matrix_Mul (A[0...n-1, 0...n-1], B[0...n-1, 0...n-1],
//Problem Description : This algorithm performs multiplication
//of two square matrices
//Input : Two matrices A and B
//Output : C matrix containing multiplication of A and B
        for i←0 to n-1 do
            for j←0 to n-1 do
                C[i, j] ← 0
                    for k ← 0 to n-1 do
                        C[i,j] ← C[i,j] + A[i,k]*B[k,j]
        return C
```

Mathematical Analysis

Step 1 : The input size of above algorithm is simply order of matrices i.e. n.

Step 2 : The basic operation is in the innermost loop and which is

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

We should note that in this basic operation both addition and multiplication are performed. But we will not choose any one of them as basic operation because on each repetition of innermost loop, each of the two will be executed exactly once. So by counting one automatically other will be counted. Hence we consider multiplication as a basic operation.

Step 3 : The basic operation depends only upon input size. There are no best case, worst case and average case efficiencies. Hence now we will go for computing sum. There is just one multiplication which is repeated on each execution of innermost loop. (a for loop using variable k). Hence we will compute the efficiency for innermost loops.

Step 4 : The sum can be denoted by M(n).

$\therefore$ M(n) = Outermost loop × Inner loop × Innermost loop (1 execution)

= [For loop using i] × [For loop using j]

× [For loop using k] (1 execution)

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} n \right)$$

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

$$M(n) = n^3$$

$\sum_{j=1}^{n-1} = n$

Thus the simplified sum is n^3. Thus the time complexity of matrix multiplication $\Theta(n^3)$.

4. Counting number of bits in an integer.

```
Algorithm Binary(n)
//Problem Description: This algorithm is for counting the
//binary digits from a decimal integer
//Input : The decimal integer n
//Output : Returns total number of digits from the input
 count ← 1
 while(n>1)
{
     count ← count+1
        n ←⌊ n/ 2 ⌋
}
return count
```

Mathematical Analysis

Step 1 : The input size is n i.e. the positive integer whose binary digits in binary representation needs to be checked.

Step 2 : The basic operation is denoted by **while** loop. And it is each time checking whether n > 1. The **while** loop will be executed for the number of time at which n > 1 is true. It will be executed once more when n > 1 is false. But when n > 1 is false the statements inside **while** loop won't get executed.

Step 3 : The value of n is halved on each repetition of the loop. Hence efficiency of algorithm is equal to $\log_2 n$.

Step 4 : Hence total number of times the **while** loop gets executed is

$$\lfloor \log_2 n \rfloor + 1$$

Hence time complexity for counting number of bits of given number is $\Theta(\log_2 n)$. The $\lfloor\ \rfloor$ indicates floor value of $\log_2 n$.

Example 1.12.1 *Show how to implement a stack using two queues. Analyze the running time of the stack operations.* **AU : Dec.-15, Marks 10**

Solution : The implementation of stack involves two operations – i) push operation ii) pop operation.

Push operation : The push operation is simple. Just insert the elements one by one to Q1 (i.e. Queues).

Pop operation : Step 1 : One by one delete the elements from Q1 to Q2 except the last element.

Step 2 : Delete the last element of Q1 and return it as popped element.

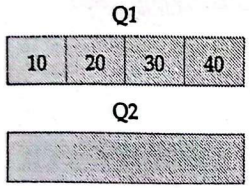
Step 3 : Interchange the names of Q1 and Q2. Repeat step 1 to step 3

Step 4 : Finally last element is deleted from Q1 and returned.

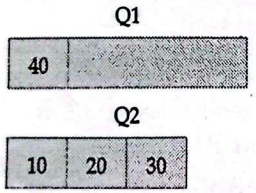
For example - consider following sequence of operations -

push 10, push 20, push 30, push 40, POP, POP, POP, POP.

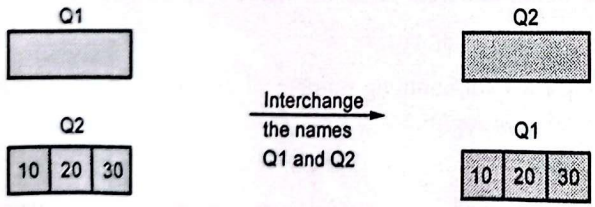
Step 1 : Push 10, 20, 30 and 40 one by one to Q1.



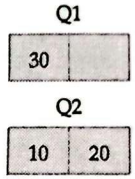
Step 2 : POP For this operation transfer all the elements one by one from Q1 to Q2 except the last element.



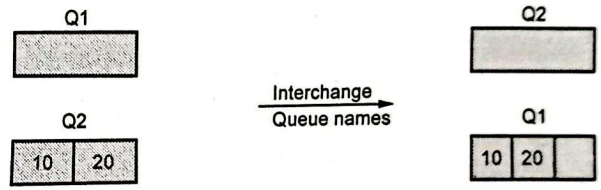
Delete 40 and return it as **pooped value**. Then interchange the queue names.



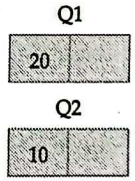
Step 3 : POP for this operation transfer all the elements except the last element



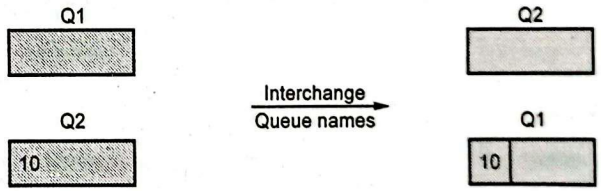
Delete 30 and return it as **popped value**. Then interchange the names of the queues.



Step 4 : POP for this operation transfer all the elements except the last element from Q1 to Q2 except the last element.



Delete 20 (from Q1) and return it as popped value. Then interchange the names of the queues.



Step 5 : Finally delete element from Q1 and return it as popped value i.e. **10**. Thus we get 40, 30, 20, 10 as popped values.

As both the queues are empty we terminate the execution of the procedure.

Analysis : The push operation take O(1) time.

The pop operation takes O(n) time as it takes (n–1) elements to transfer from one queue to another.

Example 1.12.2 *Write an algorithm for insertion sort and analyse it.* **AU : Dec.-15, Marks 8**

Solution :

```
Algorithm Insert_sort(A[0...n-1])
//Problem Description: This algorithm is for sorting the
//elements using insertion sort
//Input: An array of n elements
//Output: Sorted array A[0...n-1] in ascending order
  for i ← 1 to n-1 do
{
          temp ← A[i]//mark A[i]th element
          j ← i-1//set j at previous element of A[i]
          while(j>=0)AND(A[j]>temp)do
          {
     //comparing all the previous elements of A[i] with
     //A[i].If any greater element is found then insert
     //it at proper position
          A[j+1] ← A[j]
          j ← j-1
          }
          A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis

When an array of elements is almost sorted then it is **best case** complexity. The best case time complexity of insertion sort is **O(n).**

If an array is randomly distributed then it results in **average case** time complexity which is $O(n^2)$.

If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

University Questions

1. *Give the algorithm to check whether all the elements in a given array of n elements are distinct. Find worst case complexity of the same.* **AU : May-16, Marks 8**
2. *State the general plan for analyzing the time efficiency of nonrecursive algorithms and explain with an example.* **AU : Dec.-16, Marks 8**
3. *Briefly explain the mathematical analysis of recursive and non-recursive algorithm.* **AU : May-17, Marks 13**

1.13 Visualization

Definition: *Algorithmic visualization is a technique in which images are used to convey the information about the algorithm.*

- In algorithmic visualization the operations should be illustrated with some visual components or by some animations.
- The behaviour of algorithm on different inputs should be highlighted. Not only this, it is always effective to represent the comparison made with different algorithms solving same problem.
- There are **two approaches** of algorithmic visualization.

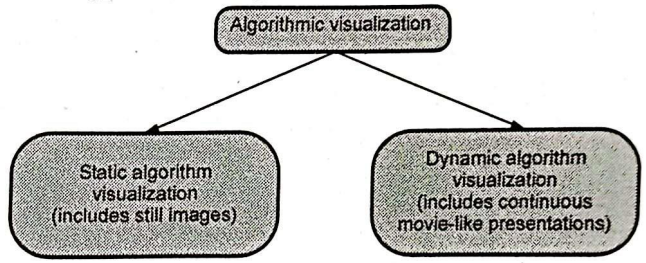
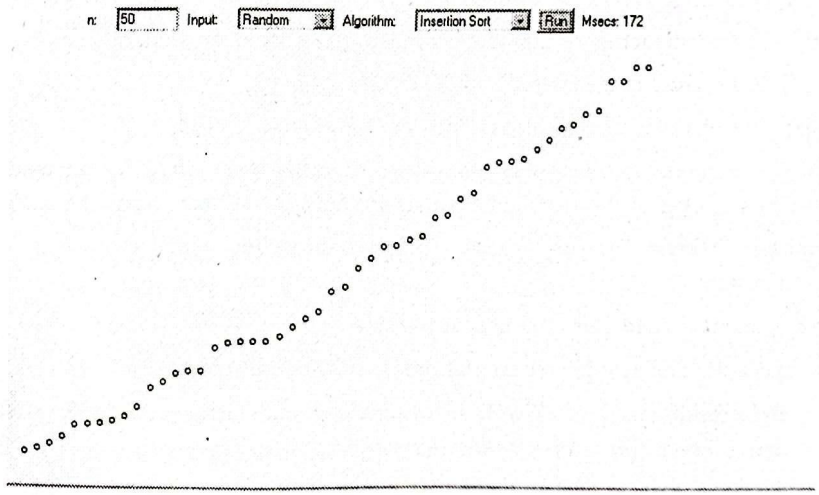
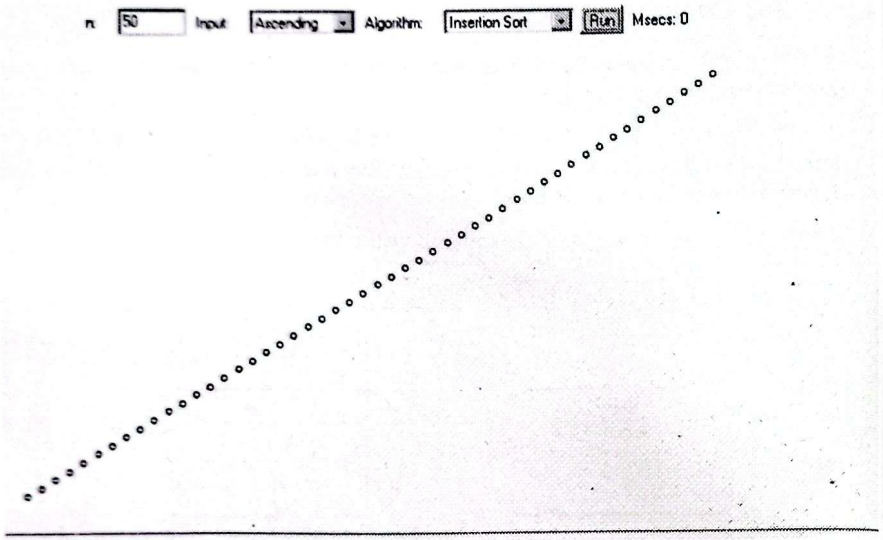


Fig. 1.13.1

- The dynamic representation of algorithmic visualization is called **algorithm animation.**
- For example -The following snapshot shows insertion sort method for random input set of 50. The time shown in Msec also shows the time taken by the algorithm to execute.



Similarly the elements can be sorted in ascending order by insertion sort method. It illustrated by following snapshot.



Features of algorithmic visualization

While showing the algorithm by animations, the following features should adopted by the developed system :

1. The visualization should be **consistent.**
2. It should be **interactive** so that any ordinary user should understand it easily.
3. It should be **clear and concise.**
4. There should be **user friendliness** with the developed system.
5. While developing such visualization first the developer should understand **knowledge level** of the user and then accordingly the system should be designed
6. **Emphasis** should be on **visual component** rather than producing textu information.
7. Such systems should keep the user **interested.**
8. The **symbolic and iconic representations** should be incorporated.
9. The **algorithmic analysis** as well as comparison with different algorithms solvi the same problem should be included in such animated systems.
10. The **execution history** should be included in such systems.

Applications of algorithm visualization

1. Research

- Many uncovered features of algorithm can be effectively shown by visual systems and researcher can get the benefit of such systems for further development and studies.

2. Education

- Students can learn algorithms very easily using animated systems. Various visual components included in the animated systems help the student to understand and analyze the algorithm in simplest manner. For example different sorting algorithms can be analyzed and understood with the help of bar charts or line graphs.

Two Marks Questions with Answers

Q.1 Formally define the notion of algorithm with diagram. AU : Dec.-09, May-13,17

Ans. : **Definition of 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.

Refer Fig. 1.1.1.

Q.2 What are six steps processes in algorithmic problem solving ? AU : Dec.-09

Ans. : The steps in algorithmic problem solving are -

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

Q.3 What do you understand by the term algorithmic strategy ?

Ans. : **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.

Q.4 Enlist few algorithmic strategies.

Ans. : 1. Brute Force 2. Divide and Conquer 3. Greedy Method
4. Dynamic Programming 5. Backtracking 6. Branch and Bound.

Q.5 What are the drawbacks in using the standard unit of time, to measure the runtime of an algorithm ? AU : Dec.-09

Ans. : The runtime of an algorithm can not be measured in standard unit of time because in multiuser system running time depends on many factors such as -

1) System load 2) Number of other programs running

3) Instruction set used 4) Speed of underlying hardware

If we use standard unit of time such as seconds, milliseconds, minutes and so on for measuring the runtime of algorithm then we get varied result for same set of input data. Hence the time complexity given in terms of **frequency count.**

Q.6 What are the measures that are used to measure the complexity of an algorithm ?

Ans. : The time complexity and the space complexity are the two measures that are used to measure the complexity of an algorithm.

Q.7 Write the concept of time and space complexity. AU : Dec.-12

Ans. : Time complexity is the amount of time required by an algorithm to execute. For computing the time complexity the frequency count of the basic operation in the algorithm is computed. This frequency count is then denoted in terms of asymptotic notations to express the time complexity of an algorithm. The space complexity is an amount of space required by an algorithm. The space complexity is denoted in terms of asymptotic notations.

Q.8 Differentiate time complexity from space complexity. AU : May-10

Ans. : Time complexity is amount of time required by a program to execute.

The space complexity is amount of space required by a program to execute.

Q.9 What is recurrence equation ? AU : May-10, Dec.-16

Ans. : The recurrence equation is an equation that defines a sequence recursively. It is normally in following form -

$$T(n) = T(n-1) + n \quad \text{for } n > 0 \qquad \ldots (1)$$

$$T(0) = 0 \qquad \ldots (2)$$

Here equation (1) is called **recurrence relation** and equation 2 is called **initial condition.** The recurrence equation can have infinite number of sequences. The **general solution** to the recursive function specifies some formula.

Q.10 Establish the relation between O and Ω . AU : Dec.-10

Ans. : The notation O represents the upper bound of algorithm's running time. The omega notation Ω represents the lower bound of algorithm's running time.

The theta notion represents the running time between upper and lower bound. If f(n) and g(n) are two functions then, $f(n) = \Theta(n)$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Q.11 If $f(n) = a_m n^m + \ldots a_1 n + a_0$, then prove that $f(n) = O(n^m)$.

Ans. : Let,

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + a_{m-2} n^{m-2} + \ldots + a_0 \qquad \ldots (1)$$

If we treate a_m^i as a constant then the equation (1) becomes

$$f(n) = A \sum_{i=0}^{m} a^m \qquad \ldots (2)$$

where A is $a_0^0 + a_1^1 + a_2^2 + \ldots$

If equation (2) is

$$f(n) = A \left[\sum_{i=0}^{n} 1 \right] = A [1 + 1 + 1 + \ldots + 1] = A\left(n^1\right)$$

$$= A \cdot O\left(n^1\right)$$

If equation (2) is

$$f(n) = A \sum_{i=0}^{n} n = A [1 + 2 + 3 + \ldots n] = A\left(n^2\right)$$

$$= A \cdot O\left(n^2\right)$$

If equation (2) is

$$f(n) = A \sum_{i=0}^{n} n^2 = A\, O\left(n^3\right)$$

Thus $$f(n) = A \sum_{i=0}^{m} n = A\, O\left(n^m\right)$$

Thus neglecting the constant term we will have

$$f(n) = O\left(n^m\right)$$

Q.12 Using the step count method analyze the time complexity when 2 m × n matrices are added. AU : May-11

Ans. : For computing the step count for the matrix addition, consider the following code -

```
1.  for (i ← 0; i < m; i++)
2.  {
3.      for (j ← 0; j < n; j++)
4.      {
5.          C [i][j] = A[i][j] + B[i][j]
6.      }
7.  }
```

Statement	Step count
i ← 0	1
i < m	m + 1
i ++	m
j ← 0	m
j ← n	(n + 1) m
j ++	(n + 1) m
C[i][j] = A[i][j] + B[i][j]	n(m)
Total	3mn + 5m + 2

By neglecting constants, the time complexity = **O(mn)**

Q.13 An array has exactly n nodes. They are filled from the set {0, 1, 2, ..., n – 1, n}. There are no duplicates in the list. Design an O(n) worst case time algorithm to find which one of the elements from the above set is missing in the array. AU : May-11

Ans. : Algorithm : GetMissing (Array a[0..n])

```
{
total = (n + 1) * (n + 2)/2//Finding sum of all the elements from 0 to n
        for (i ← 0 to n) do
        total – = a[i]          //subtracting each number from total
        return total
}
```

Remaining will be missing element

Q.14 What do you mean by order of growth ?

Ans. : Measuring the performance of an algorithm in relation with the input size n is called order of growth.

Q.15 What is the best,worst and average case time complexities of linear search algorithm ?

Ans. : The O(n) is the best,worst and average case time complexity of linear search algorithm.

Q.16 What is time space tradeoff ?

Ans. : Time space tradeoff is basically a situation where either a space efficiency (memory utilization) can be achieved at the cost of time or a time efficiency (performance efficiency) can be achieved at the cost of memory.

Q.17 Give any two examples explaining the time space tradeoff.

Ans. : **Example 1 :** Consider the programs like compilers in which symbol table is used to handle the variables and constants. Now if entire symbol table is stored in

the program then the time required for searching or storing the variable in the symbol table will be reduced but memory requirement will be more. On the other hand, if we do not store the symbol table in the program and simply compute the table entries then memory will be reduced but the processing time will be more.

Example 2 : Suppose, in a file, if we store the uncompressed data then reading the data will be an efficient job but if the compressed data is stored then to read such data more time sill be required.

Q.18 Define Big oh notation. AU : May-12

Ans. : Let f(n) and g(n) be two non-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$f(n) \leq c * g(n)$$

then f(n) is big oh of g(n). It is also denoted as $f(n) \in O(g(n))$. In other words f(n) is less than g(n) if g(n) is multiple of some constant c.

Q.19 Define Big omega notation. AU : May-13

Ans. : A function f(n) is said to be in $\Omega(g(n))$ if f(n) is bounded below by some positive constant multiple of g(n) such that

$$f(n) \geq c * g(n) \qquad \text{For all } n \geq n_0$$

It is denoted as $f(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.

Q.20 Define Big theta notation. AU : Dec-14

Ans. : Let F(n) and g(n) be two non negative functions. There are two positive constants namely c_1 and c_2 such that

$$c_1 \leq g(n) \leq c_2\, g(n)$$

Then we can say that

$$f(n) \in \Theta(g(n))$$

Q.21 Write any two properties of big-oh notation. AU : Dec-11

Ans. : Following are some important properties of big oh notations -

1. If there are two functions $f_1(n)$ and $f_2(n)$ such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = \max(O(g_1(n)), O(g_2(n)))$.
2. If there are two functions $f_1(n)$ and $f_2(n)$ such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$.

Q.22 Is big-Oh transitive ?

Ans. : Yes. Because if f(n) = O(g(n)) and g(n)-O(h(n)) then f(n) = O(h(n))

Q.23 List the asymptotic classes in increasing order.

Ans. : 1. Constant(1) 2. Logarithmic(log n) 3. Linear(n) 4. n log n
5. Quadratic (n^2) 6. cubic(n^3) 7. Exponential(2^n) 8. Factorial(n!)

Q.24 Define Breakeven point.

Ans. : The break even point is some value of n after which the larger order of magnitude dominates. For example : The complexity of $c_1n^2+c_2n$ is worse than $c_3n \log n$ for larger values of n.

Q.25 What is conditional asymptotic notation ?

Ans. : Many algorithms are easier to analyse if we impose conditions on them initially. Imposing such conditions, when we specify asymptotic value is called **conditional asymptotic notation.**

Q.26 Define algorithm validation. **AU : Dec.-12**

Ans. : The process of measuring the effectiveness of the algorithm before actually making program or code from it, in order to know whether the algorithm is correct for valid input is known as algorithm validation. Algorithm validation can be done with the help of mathematical and empirical methods.

Q.27 What are the components of fixed and variable part in space complexity ? **AU : Dec.-13**

Ans. : The space requirement S(p) is given by following formula -

$S(p)=C+Sp$

Where C is a constant or fixed part that denotes the space of inputs and outputs. This space is the amount of space taken by instructions, variables and identifiers.

The Sp denotes the variable part of the space. It denotes the space requirement based on particular problem instance.

Q.28 What is average case analysis ? **AU : May-14**

Ans. : In average case analysis, all the possible inputs are considered and the computing time for all of the inputs is calculated. The sum of all the calculated values is then divided by total number of inputs. The value which we obtain by this computation yields the average case analysis.

Q.29 Define program proving and program verification **AU : May-14**

Ans. : Program proving means proving each and every instruction of the program with the help of mathematical theorems. Program verification means checking the correctness of the program.

Q.30 What is meant by substitution method ? **AU : Dec.-14**

Ans. : Substitution method is used for solving the recurrence relation. It is a kind of method in which a guess for the solution is made. There are two types of substitutions made -

1. Forward substitution method 2. Backward substitution method

Q.31 Write down algorithm to find the number of binary digits in the binary representation of a positive decimal integer. **AU : May-15**

Ans. :

```
Algorithm CountBinDigits(int dec)
{
    int bin[size];
    int i,zeros,ones,total_digits;
    zeros=ones=i=0;
    /*Reading the decimal Number*/
    Write("Enter the decimal number to find its binary number\n");
    Read(dec);
    while(dec>0) do
    {
        bin[i]=dec%2;
        i++;
        dec=dec/2;
    }
    Write("\nBinary number is ");
    for(int j=i-1;j>=0;j--)do
    {
        Write(bin[j]);
        if(bin[j]==0)
            zeros++;
        else
            ones++;
    }
    Write("\n Number of zeros are ",zeros);
    Write("\n Number of ones are ",ones);
    total_digits=zeros+ones;
    Write("\n Total Number of Binary Digits are ",total_digits);
}
```

Q.32 Write down the properties of asymptotic notations. **AU : May-15**

Ans. : Properties of asymptotic notations

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

Let, n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

1. $f(n) \leq c*g(n)$

   Then f(n) is said to be big oh of g(n).

   i.e. $f(n) \in 0\ (g(n))$

2. $f(n) \geq c*g(n)$ for all $n \geq n_0$

   Then function f(n) is said to be big omega of g(n).

   i.e. $f(n) \in \Omega\ (g(n))$.

3. Let there be two positive constants namely C_1 and C_2 such that

   $C_1*g(n) \leq f(n) \leq C_2*g(n)$

The function f(n) is said to be big theta of g(n).

i.e. $f(n) \in \Theta(g(n))$.

Q.33 The (log n)th smallest number of n unsorted numbers can be determined in O(n) average, case time (True / False) AU : Dec.-15

Ans. : False. The (log n)th smallest number of n unsorted numbers can be determined in O(n) worst-case time.

Q.34 Write the recursive Fibonacci algorithm and its recurrence relation. AU : Dec.-15

Ans. : Recursive Fibonacci Algorithm

Algorithm Fib(n)

```
{   if(n < = 1) then
    return n ;
    else
        return(Fib(n – 1) + Fib(n – 2)) ;
}
```

The recurrence relation would be

$$T(n) = T(n-1) + T(n-2)$$

$$T(0) = 0$$

$$T(1) = 1$$

Q.35 Compare the orders of growth of n(n–1)/2 and n^2. AU : May-16

Ans. : For comparing the orders of growth of n(n–1)/2 and n^2 we consider some sample values of n.

n	$\frac{n(n-1)}{2}$	n^2
1	0	1
2	1	4
3	3	9
4	6	16
5	10	25

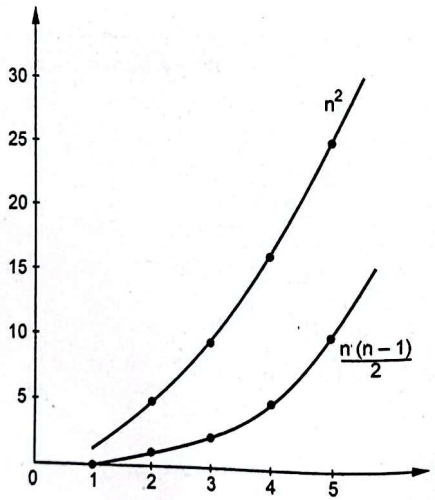


Fig. 1.1 Order of growth

Q.36 What is called basic operation of an algorithm ? AU : May-18

Ans. : The operation which is more time consuming is the basic operation. For example - For searching a key element from the list, the basic operation is comparison of key with every element of the list.

Q.37 Give the time complexity 1 + 3 + 5 + 7 + ... + 999.

Ans. : $1 + 3 + 5 + 7 + \ldots + 999 = \sum_{i=1}^{500}(2i-1)$

$$= \sum_{i=1}^{500} 2i - \sum_{i=1}^{500} 1 = 2 * \frac{500*501}{2} - 500$$

i.e. $\sum_{i=1}^{500} 2i-1 = i^2 = 2{,}50{,}000$

Hence time complexity is $O(n)^2$.

Q.38 Give the Euclid's algorithm for computing gcd(m, n). AU : May-16 , 18

Ans. :

```
Algorithm GCD(m, n)
    {
        while (n ≠ 0) do
        {
            z = m mod n
            m = n
            n = z
        }
        return m ;
    }
```

Q.39 Write an algorithm to compute the greatest common divisor of two numbers. AU : May-17

Ans. : Refer Q.38.

Q.40 Design an algorithm to compute the area circumference of a circle. AU : Dec.-16

Ans. :

```
Algorithm compute (radius)
    {
        Pi = 3.14;
        Area = Pi*radius*radius;
        Circum = 2*Pi*radius;
        Write (Area);
        Write (Circum);
    }
```

Q.41 How to measure an algorithm's running time ? AU : Dec.-17

Ans. : To measure the algorithm's running time, its basic operation is identified. Then the frequency count of this basic operation is calculated. And finally applying the asymptotic notations for basic operations, the time complexity is calculated.

Q.42 What do you mean by "Worst case - efficiency" of an algorithm ? AU : Dec.-17

Ans. : Worst case time complexity is a time complexity when algorithm runs for a longest time.

Q.43 State Master's theorem. AU : May-18

Ans. :

1. If $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then
   $T(n) = \Theta(n^{\log_b a})$
2. If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then
   $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then
   $T(n)$ is $= \Theta(f(n))$

❑❑❑

UNIT - II

2 Brute Force and Divide and Conquer

Syllabus

Brute Force - Computing a^n - String Matching - Closest-Pair and Convex-Hull Problems - Exhaustive Search - Travelling Salesman Problem - Knapsack Problem - Assignment problem. Divide and Conquer Methodology - Binary Search - Merge sort - Quick sort - Heap Sort - Multiplication of Large Integers - Closest-Pair and Convex - Hull Problems.

Contents