UNIT 1 BASICS OF AN ALGORITHM AND ITS PROPERTIES

1.0 INTRODUCTION

Studying algorithms is an exciting subject in computer science discipline. We come across a large number of interesting problems and techniques to solve to solve these problems. Not every problem can be solved with the existing techniques but majority of them can be. But let us define what is an algorithm first. The word *algorithm* is derived from the mathematician of the ninth century *Abdullah Jafar Muhammad ibn Musa Al-khowarizmi*. The word 'al-khowarizmi' is 'Algorismus'in Latin which became Algorithm after his name.

He defined Algorithm as:

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a well-defined computational procedure that takes input and produces output.
- An algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve some particular output.

In this unit, the basics of the algorithms and its designing process will be discussed. Section 1.3 will define the algorithm and its uses with suitable example. An algorithm is designed with five basic building blocks, namely sequencing, selection, and iteration. A detailed discussion about these building blocks of an algorithm is presented in Section 1.4.

The solution of a problem can be achieved through a number of algorithms. To check which algorithm is better than the others, a parameter, known as time complexity, is used. Therefore, time complexity is one of the important concepts related to algorithm which are discussed in Section 1.5. Section 1.6 deals with the analysis of Algorithms. To compare Algorithms, complexity is the parameter to be considered. Computing problems are categorized according to their solving approach. These are discussed in section 1.7. Section 1.8 comprises the solving techniques of various computing problems. In section 1.9 Deterministic and Stochastic Algorithm are discussed. An algorithm is deterministic if the next output can be predicted/ determined from the input and the state of the program, whereas stochastic algorithms are random in nature.

Chapter is summarized in section 1.10. In section 1.11 review questions of the chapter are covered for check pointing purpose. In Section 1.12 a list of reference material is enlisted for further readings.

1.1 OBJECTIVES

After studying this unit, you should be able to:

- Define and list properties of an algorithm.
- List basics building blocks of algorithms.
- Explain fundamental techniques to design an algorithm.
- Define the time and space complexity of an algorithm.
- Differentiate between deterministic and stochastic algorithms.

1.2 EXAMPLE OF AN ALGORITHM

An algorithm is not a coding instruction rather it is a sequence of tasks written in common language, if executed produces certain output within a time frame. An algorithm is completely independent of programming language.

For a good algorithm, it must satisfy the following characteristics or properties:

1. **Input**: There must be a finite number of inputs for the algorithm.
2. **Output**: There must be some output produced as a result of execution of the algorithm.
3. **Definiteness**: There must be a definite sequence of operations for transformation of input into output.
4. **Effectiveness**: Every step of the algorithm should be basic and essential.

5. **Finiteness**: The transformation of input to output must be achieved in finite steps, i.e. the algorithm must stop, eventually! Stopping may mean that it should produce the expected output or a response that no solution is possible.

Following are desirable characteristics of an algorithm:

- The **algorithm should be general** and is able to solve several cases.
- The **algorithms should use resources efficiently**, i.e. takes less time and memory in producing the result.
- The **algorithms should be understandable** so that anyone can understand and apply it to own problem.
- The **algorithm should follow the uniqueness** such that each instruction of the algorithm is unambiguous and clear.

The "analysis" of an algorithm finds the suitability of it in terms of the time and space (memory) complexity also known as the performance evaluation of the algorithm.
Before proceeding further to discuss about writing the algorithm and its analysis, let's first see how an algorithm looks like. For the same, let us consider a well-known algorithm to find the Greatest Common Divisor (GCD) of two given integers. We can write an algorithm in any natural language. Here, we are presenting the algorithm in English language.

To find GCD of two given Integers, we are using an efficient **Euclid's algorithm** which is named after the ancient Greek mathematician Euclid.

The pseudo code for computing GCD (a, b) by Euclid's method is as follows:

// a and b are two positive numbers where a is dividend and b is a divisor

1. If b=0, return a and exit
2. else go to step 3
3. Divide a by b and assign remainder to r
4. Assign the value of b to a and the value of r to b and go back to step 1

To validate the algorithm, it must produce the desired result within finite number of steps. The above-mentioned algorithm has two inputs and one output. The algorithm is also definiteness and written in basic and effective sentences. The algorithm is also finite as it terminates in finite steps. To observe the same, let us find the GCD of $\boldsymbol{a}$ = 1071 and $\boldsymbol{b}$ = 462 using Euclid's algorithm.

Iteration 1:

1. Divide **a**=1071 by **b**=462 and store the remainder in **r**.

   r= 1071 % 462 (here, % represents the remainder operator)

r = 147

2. If **r** = 0, the algorithm terminates and **b** is the GCD. Otherwise, go to Step 3.

   Here, **r** is not zero, so we will go to Step 3.

3. The integer will get the current value of integer **b** and the new value of integer **b** will be the current value of **r**.

   Here, **a**=462 and **b**=147

4. Go back to Step 1.

Iteration 2:

1. Divide **a**= 462 by **b**= 147 and store the remainder in **r**.

   r= 462 % 147 (here, % represents the remainder operator)

   r = 21

2. If **r** = 0, the algorithm terminates and **b** is the GCD. Otherwise, go to Step 3.

   Here, **r** is not zero, so we will go to Step 3.

3. The integer **a** will get the current value of integer **b** and the new value of integer **b** will be the current value of **r**.

   Here, **a**= 147 and **b**= 21

4. Go back to Step 1.

Iteration 3:

1. Divide **a**=147 by **b**=21 and store the remainder in **r**.

   r= 147 % 21 (here, % represents the remainder operator)

   r = 0

2. If **r** = 0, the algorithm terminates and **b** is the GCD. Otherwise, go to Step 3.

   Here, **r** is zero, so the algorithm terminates and b is the answer, i.e. 21.

Let us write the **Euclid's Algorithm**

```
Algorithm GCD-Euclid (a, b)
begin [start of Algorithm]
{
        while b ≠ 0 do
        {
                r ← a mod b;
                a ←b;
                b← r;
        } [end of while loop]
return (b)
}  [end of algorithm]
```

Algorithm 1: Finding GCD of (a, b) with Euclidian Method.

1.3 BASICS BUILDING BLOCKS OF ALGORITHMS

An algorithm is a procedural way to write the solution of a problem. It is designed with five basic building blocks, namely: sequencing, selection, iteration procedure and recursion. In the first subsection we discuss sequencing, iteration and selection and in the subsequent subsection procedure & recursion will be explained.

Sr. N.	Building Block	Common Name
1.	Sequencing	Step by step actions
2.	Selection	Decision
3.	Iteration	Repetition or Loop
4.	Procedure	
5.	Recursion	

1.3.1 Sequencing, Selection and Iteration

A solution of a problem mainly comprises these three basic blocks only. In an Algorithm there may be actions to be performed linearly or sequentially as they written in text. At some times the next action/ statement to be executed is

decided based on some condition called the selection of next action to be performed. Some set of actions/statements are to be executed more than once called repetition or the loop.

Let's consider the example of finding GCD of (a, b) with Euclidian method to understand the basic building blocks of an algorithm(Algorithm 1)

Sequencing: A problem can be solved by performing some actions in a sequence [called algorithm], and the order of execution of those actions is important to ensure the correctness of an algorithm.

If the order of steps of algorithm changes and does not follow the steps as specified, it will not produce the correct output as expected.

Selection: As an algorithm has to be generalized to solve many cases, there may be situations where the sequence of execution of actions may depend on some condition. That is, some of the instructions will only be executed if a given condition satisfies.

It is like the next action to be performed is dependent upon some Boolean expression. So, using selection, next step to be executed is determined.
Iteration: While solving a problem certain actions may be required to execute a certain number of times or until a certain condition is met.

Let us consider Algorithm 1 again to understand these concepts: Finding GCD of (a, b) with Euclidian Method.

Algorithm GCD-Euclid (a, b)

Step1: begin [start of Algorithm]

Step2: {

Step3: while b ≠ 0 do

Step4: {

Step5: r ← a mod b;

Step6: a ←b;

Step7: b← r;

Step8: } [end of while loop]

Step9: return (b)

Step10: } [end of algorithm]

In above algorithm Step5, Step6 and Step7 are example of sequencing, as these statements are always executed in sequence as written the text.

Step3 is selection step as it decides which next step is to be executed among Step4 and Step9 according to the condition of the loop.

Step3 is also acts as iteration or the looping statements. Based on the while loop condition, the Step4 to Step8 are executed in repeatedly manner.

1.3.2 Procedure & Recursion

Though the above-mentioned three control structures, viz., direct sequencing, selection and repetition, are sufficient to express any algorithm, yet the following two advanced control structures have proved to be quite useful in facilitating the expression of complex algorithms viz.

(i) Procedure
(ii) Recursion

Let us first take the advanced control structure *Procedure*.

Procedure

Among a number of terms that are used, instead of procedure, are subprogram and even function. These terms may have shades of differences in their usage in different programming languages. However, the basic idea behind these terms is the same, and is explained next.

It may happen that a sequence frequently occurs either in the same algorithm repeatedly in different parts of the algorithm or may occur in different algorithms. In such cases, writing repeatedly of the same sequence, is a wasteful activity. *Procedure* is a mechanism that provides a method of checking this wastage. For example we can define GCD(a, b) as a procedure/function only once and can call it a number of times in a main function with different values of a and b

Under this mechanism, the sequence of instructions expected to be repeatedly used in one or more algorithms, is written only once and outside and independent of the algorithms of which the sequence could have been a part otherwise. There may be many such sequences and hence, there is need for an identification of each of such sequences. For this purpose, each sequence is prefaced by statements in the following format:

Procedure <Name> (<parameter-list>) [*: < type>*]
<declarations>
<sequence of instructions expected to be occurred repeatedly>
end;

In cases of procedures which pass a value to the calling program another basic construct (in addition to *assignment, read and write*) viz., *return (x)* is used, where x is a variable used for the value to be passed by the procedure.

There are various mechanisms by which values of *a and b* are respectively associated with or transferred to *x and y*. *The variables like a and b*, defined in the calling algorithm to pass data to the procedure (*i.e., the called algorithm*), which the procedure may use in solving the particular instance of the problem, are called **actual parameters** or **arguments**.

Recursion

Next, we consider another important control structure namely recursion. In order to facilitate the discussion, we recall from Mathematics, one of the ways

in which the factorial of a natural number n is defined:

factorial (1) = 1

factorial (n) = n*factorial(n─1).

For those who are familiar with recursive definitions like the one given above for factorial, it is easy to understand how the value of (n!) is obtained from the above definition of factorial of a natural number. However, for those who are not familiar with recursive definitions, let us compute factorial (4) using the above definition.

By definition
factorial (4) = 4 * factorial (3).
Again by the definition
factorial (3) = 3 * factorial (2)
Similarly
factorial (2) = 2* factorial (1)
And by definition
factorial (1) = 1

Substituting back values of factorial (1), factorial (2) etc., we get factorial (4) = 4.3.2.1=24, as desired.

This definition suggests the following procedure/algorithm for computing the factorial of a natural number n:

In the following procedure factorial (n), let *fact* be the variable which is used to pass the value by the procedure *factorial* to a calling program. The variable fact is initially assigned value 1, which is the value of factorial (1).

Procedure factorial (n)

```
    fact: integer;
begin
    fact ←1
    if n equals 1 then return fact
    else begin
    fact ←n * factorial (n –1)
    return (fact)
    end;
end;
```

In order to compute *factorial (n ─1)*, procedure *factorial* is called by itself, but this time with (simpler) argument (n ─1). The repeated calls with simpler arguments continue until factorial is called with argument 1. Successive multiplications of partial results with 2,3, …..up to n finally deliver the desired result.

Definition: A procedure, which can call itself, is said to be **recursive procedure/algorithm**. For successful implementation of the concept of recursive procedure, the following conditions should be satisfied.

(i) There must be in-built mechanism in the computer system that supports the calling of a procedure by itself, e.g, there may be in-built stack operations on a set of stack registers.

(ii) There must be conditions within the definition of a recursive procedure under which, after finite number of calls, the procedure is terminated.

(iii) The arguments in successive calls should be simpler in the sense that each succeeding argument takes us towards the conditions mentioned in (ii).

Recursion is an important construct which will be used extensively to solve sorting algorithms, searching algorithm, matrix multiplications, etc.

1.4 A SURVEY OF COMMON RUNNING TIME

For a given problem, more than one algorithm can be designed. However, one algorithm may be better than the other. To compare two algorithms for a problem, running time is generally used which is defined as the time taken by an algorithm in generating the output. An algorithm is better if it takes less running time. The "time" here is not necessarily the clock time. However, this measure should be invariant to any hardware used. Therefore, the running time of an algorithm can be represented in terms of the number of operations executed for a given input. More the number of operations, the larger the running time of an algorithm. So, if we can find the number of operations required for a given input in an algorithm then we can measure the running time. This running time of an algorithm for producing the output is also known as **time complexity**. Time here is not the clock time

Therefore, two algorithms can be compared in terms of time complexity. an Algorithm is better compared to others having smaller running time (time complexity).

Running time of an algorithm is represented as a function $T(n)$, where n is the input size. Let, an algorithm has a running time $T(n) = cn$, where c is a constant. The running time for this algorithm is linearly dependent on the size of the input. The unit of $T(n)$ is unspecified.

Following are the generalized form of running time for the algorithms:

1. **Constant Time($O(1)$):** If the running time does not depend on the input size (n) then it is known as constant running time. It can be represented as

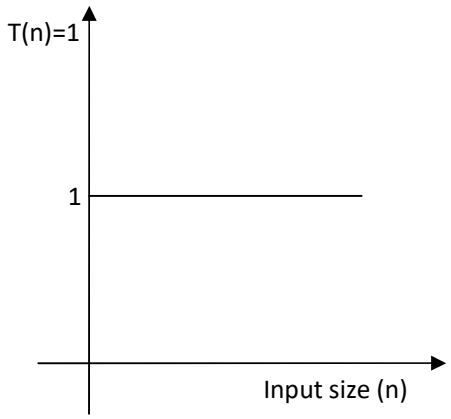


Figure (a): *T(n) = O(1)*

2. **Linear Time *O(kn)*:** If the time complexity is at most a constant factor times the size of the input, then it is known as linear time complexity and is presented as *T(n) <=kn* where k is a constant or T(n) =*O(n)*. An algorithm of this type of complexity generally completes the execution in a single pass. For example to search for minimum value of *a* given *n* numbers in an array the processing can be completed just in one pass. The following program fragment demonstrates. In this way we perform constant amount of work in processing each element of an array.

```
minimum = a[1]
for í = 2 to n
if a[í] < minimum
minimum = a[í]
end
end if
```

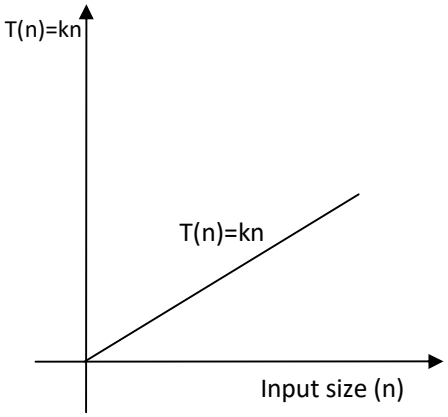


Figure (b): T(n) = O(n)

3. **Logarithmic Time(*log(n)*)**: If the time complexity of an algorithm is proportional to the logarithm of the input size, then it is known as logarithmic time complexity and depicted as *O(log n)*time. For example running time of binary search algorithm is *O(log n)*.O(n log n) is a very common running time for many algorithms which are solved through divide and conquer technique such as Merge sort ,Quick sort algorithms, etc., The common operations among all these problems are in splitting of the array in equal sized sub-arrays and then solve it recursively.

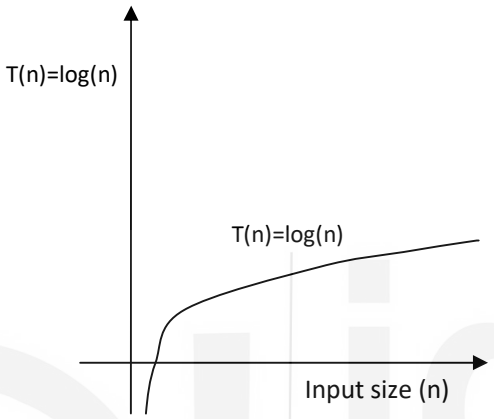


Figure (c): T(n) =O(log(n))

Quadratic Time: (*T(n)= O(n)^2)*- It occurs when the algorithm is having a pair of nested loops. The outer loop iterates *O(n)* time and for each iteration the inner loop takes *O(n)* time so we get *O(n^2)* by multiplying these two factors of *n*. Practically this is useful for problem for small input size or elementary sorting algorithms. The worst case time complexity for Bubble sort, Insertion sort, Selection sort and insertion sort running time complexities are O(n^2)

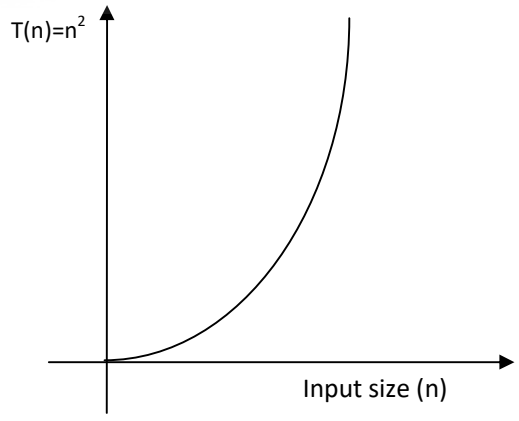


Figure (d): T(n) = O=O(n^2)

4. **Cubic Time:** (*T(n)= O(n^3)):* It often occurs when the algorithm is having there nested loops and each loop has a maximum *n* iterations. Let us have one interesting example which requires cubic time complexity. Suppose we are given n sets: S_1, S_2,….S_n. Size of each set is n (ie each set is having n elements). The problem is to find whether some pairs of these sets are disjoint, i.e there are no common elements in these pairs and what is the time complexity ?

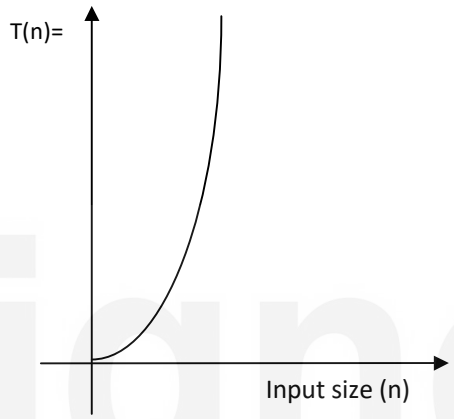


Figure (e): T(n) =O(n^3)

Pseudo-code for finding common elements in pair of sets:

```
for each set S_i of n elements
  for each other set S_j of n elements
    for each element x of S_i
      check whether x also belongs to S_j
    end for
    if x belongs to both S_i and S_j
      print "  S_j and S_j are not disjoint"
    end if
  end for
end for
```

Time Complexity- The innermost loop will be executed for n(n+1)(n+2)/6 times, which clearly states that the time complexity for this algorithm is O(n^3).

5. **Polynomial Time:** (0*(n^k)*):-This running time is obtained when the search over all subsets of a set of *a* size *k*in performed. To understand

the complexity of running time we have to find how many distinct subsets of a size *k* of *n* elements of a set can be chosen. For that we have to take a combination of n elements taken k at a time .As an example let us consider a problem to find an independent set in a graph which can be defined as a set of nodes in which no pair of nodes have an edge between them. Let us formulate the independent set problem in the following way: given a constant *k* and a graph *G* having *n* nodes (vertices) find out an independent set of a size *k*.

The brute force method to solve this problem would require searching for all subsets of *k* nodes and for each subset it would examine whether there is an edge connecting any two nodes for each subset s of a size *k* .Below is a pseudo-code for finding an independent set.

Pseudo-code

```
for each subset s of a size k in a graph G
   check whether s is an independent set
   if yes, print " s is an independent set
   else stop
```

In this case the outer loop will iterate $O(n^k)$ times and it selects all *k*-node subsets of *n* node of the graph. In the inner loop within each subset it loops for each pair of nodes to find out whether there is an edge between the pair which will require *O(* 2 out of k) pairs of search i.e. O(k^2) search. Therefore the total time now is $O(k^2n^k)$. Since k is a constant, it can be dropped, finally it is $O(n^k)$.

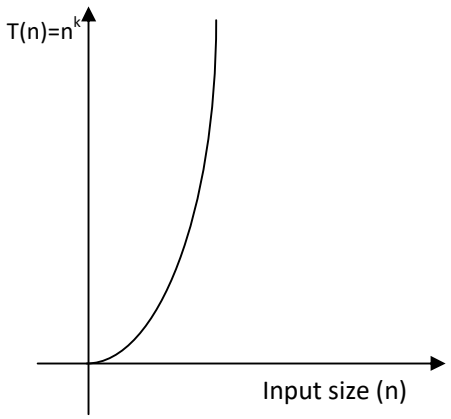


Figure (f):T(n) =O(n^k)

6. **Exponential Time:** *($O(k^n)$)* Beyond the polynomial time complexity there are other two types of bounds :exponential time *$O(2^N)$*and factorial time *O(n!):* let us refine the independent set problem that we are given a graph of a size *n* and want to find out an independent of a maximum value instead of some constant *k* which is less than *n*. The modified version of the pseudo-code is presented below.

Pseudo-code : Pseudo-code for finding an Independent Set of a graph

```
Input G(V,E)
{
for each subset s of n number of nodes
verify whether s is an independent set
if s is the largest among all the subsets examined so for
  print "s is the largest independent set "
end if
end for
}end of code fragment
```

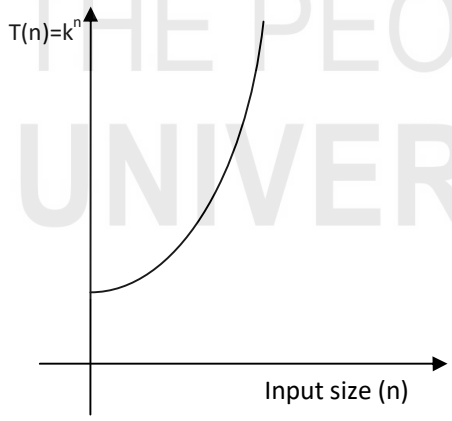


Figure (g): T(n) =O(k^n)

In this case the total number of subsets of n elements would be 2^n, so the outer loop will execute 2^n times instead of n^k times

Verification of all pairs of subsets i.e. (2^n) whether these subsets are having edges or not and then selecting the maximum will be O(n^2) i.e the total number of pair of subsets. The total running time would be

$O(n^2*2^n)$.$O(2^n)$running time complexity arises when a search algorithm considers all subsets of *n* elements.

Factorial time (O(n!):In comparison to the growth of exponential running time, the growth of factorial time (n!) is more rapid. The running time of this type of complexity arises in two types of algorithms:

(i)Algorithm for Matching, for example bipartite matching algorithm. Suppose there are *n* number boys and *n* number of girls. To find perfect matching between n number of boys &n number of girls, the first boy will be compared with *n* numbers of girls. The second boy will be left with *(n-1)* choices among girls for comparison. There will be only (n-2) options for matching for the third boy, and so forth. After array girls multiplying all these options for n boys we obtain n! ie. n(n-1) (n-2) .....(2) (1)

(ii) O(n!) also occurs where the algorithm requires arranging n elements into a particular order (i.e. a permutation of *n* numbers). A classic example is travelling salesman problem. Given a n number of cities with distance between all pairs of cities with the following conditions (i) the salesman can start the tour with any city but must conclude the tour with the starting city only (ii) all cities must be visited only once except the one where from the tour starts. The problem is to find out the shortest tour covering all n cities. Applying a brute force approach to find out the solution, a salesman has to explore n! searches which will take O(n!). Note that a salesman can pick up any city among n cities to start the tour. Next it will have (n-1) cities to pickup the second city on the tour. There will be (n-2) cities to pick up the third city at the next stage and so forth. Multiplying all these choices we get n! i.e. n (n-1) (n-2) ....(2) (1)

1.5 ANALYSIS & COMPLEXITY OF ALGORITHM

The term "analysis of algorithms" was introduced by Donald Knuth. It has become now an important computer science discipline whose overall objective is to understand the complexity of an algorithm in terms of time complexity and storage requirement.

System performance is directly dependent on the efficiency of algorithm in terms of both the time complexity as well the memory. An algorithm designed for time sensitive application takes too long to run can render its results of no use.

Suppose M is an algorithm with n the input data size. The time and space used by the algorithm M are the two main measures for the efficiency of M. The time is measured by counting the number of key operations, for example, in case of sorting and searching algorithms, the number of comparisons is the

number of key operations. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.

The ***complexity*** of an algorithm M is the *function f(n)*, which give the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. In general the term "***complexity***" given anywhere simply refers to the running time of the algorithm. There are 3 cases, in general, to find the complexity function f(n):

- Worst-case − The maximum number of steps taken on any instance of size a.
- Best-case − The minimum number of steps taken on any instance of size a.
- Average case –The number of steps taken on average for all instances of size n

The analysis of the average case assumes a certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The Average case also uses the concept of probability theory. Suppose the numbers $N_1, N_2, \ldots N_k$ occur with respective probabilities $p_1, p_2, \ldots p_k$. Then the expectation or average value of E is given by $E = N_1 p_1, N_2 p_2 \ldots . N_k p_k$

To understand the Best, Worst and Average cases of an algorithm, consider a linear array $A[1 \ldots . n]$, where the array A contains n-elements. Students may you are having some problem in understanding. Suppose you want *either* to find the location LOC of a given element (say x) in the given array A or to send some message, such as LOC=0, to indicate that x does not appear in A. Here the linear search algorithm solves this problem by comparing given x, one-by-one, with each element in A. That is, we compare with A[1], then A[2], and so on, until we find x LOC such that $x = A[LOC]$.

Algorithm: (**Linear search)**
/* **Input:** A linear list A with n elements and a searching element x.
Output: Finds the location LOC of x in the array A (by returning an index)
or return LOC=0toindicate x is not present in A.*/

1. [Initialize]: Set K=1 and LOC=0.
2. Repeat step 3 and 4 while (LOC = = 0 && K ≤ n)
3. *If (x = = A[K])*
4. {
5. LOC=K
6. K=K+1;
7. }
8. *If (LOC = = 0)*
9. *Print (" x is not present in the given array A);*
10. *Else*
11. *Print f("x is present in the given array A at location A [LOC]);*
12. *Exit [end of algorithm]*

Analysis of linear search algorithm

The complexity of the search algorithm is given by the number C of comparisons between x and array elements A[K].

Best case: Clearly the best case occurs when x is the first element in the array A. That is $x = A[LOC]$. In this case $C(n) = 1$

Worst case: Clearly the worst case occurs when x is the last element in the array A or x is not present in given array A (to ensure this we have to search entire array A till last element). In this case, we have $C(n) = n$.

Average case: Here we assume that searched element x appear in array A, and it is equally likely to occur at any position in the array. Here the number of comparisons can be anyofnumbers*1,2,3,…,n,* and each number occurs with the probability p=1/n then

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

It means the average number of comparisons needed to find the location of x is approximately equal to half the number of elements in array A. From above discussion, it may be noted that the complexity of an algorithm in the average case is much more complicated to analyze than that of worst case. Unless otherwise stated or implied, we always find and write the complexity of an algorithm in the worst case.

There are three basic asymptotic(i.e., when input size n → infinity) notations which are used to express the running time of an algorithm in terms of function, whose domain is the set of natural numbers N={1,2,3,…..}. These are:

- *0(Big – 'Oh')* [This notation is used to express Upper bound (maximum steps) required to solve a problem]
- *Ω(Big – 'Oh')* [This notation is used to express Lower bound i.e. minimum (at least) steps required to solve a problem]
- Θ ('Theta') Notations.[Used to express both Upper & Lower bound, also called tight bound]

Asymptotic notation gives the *rate of growth*, i.e. performance, of the run time for "*sufficiently large input sizes"* $(i.e.\ n \to \infty)$and is **not** a measure of the *particular* run time for a specific input size (which should be done empirically). O-notation is used to express the Upper bound (worst case); Ω-notation is used to express the Lower bound (Best case) and Θ- Notations is used to express both upper and lower bound (i.e. tight bound) on a function.

We generally want to find either or both an asymptotic *lower bound* and *upper bound* for the growth of our function.

1.6 TYPES OF PROBLEMS

The types of problems in computing are limitless, and are categorized into a few areas to make it easy for researchers to address types of problems while addressing the algorithm field.

Following are the some commonly known problem types:

- Sorting
- Searching
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

For the above-mentioned categories, certain standard input sets are defined as benchmarking sets to analyse the algorithms.

Sorting

The *sorting* is the process to arrange the given set of items in a certain order, assuming that the nature of the items allow such an ordering. For example, sorting a set of numbers in increasing or decreasing order and sorting the character strings, like names, in an alphabetical order.

Researchers have published a large number of different sorting algorithms, targeting various types of items. A sorting algorithm does not necessarily work optimally for all types of items list. Some of them are good in terms of resource usage, while some are fast in terms of computing. The efficiency of a sorting algorithm also depends on the type of input, some work well on randomly ordered inputs, whereas others perform better on already almost-sorted lists. Some of the sorting algorithms perform well for lists residing in the memory, while others perform optimally for sorting large files stored on a secondary disk.

As of now in common, the best sorting algorithm takes *nlogn* comparisons to sort an item list of n items.

For any sorting algorithm following two characteristics are desirable:

1. Stability
2. In-place.

A sorting algorithm is called stable if it does not change the relative positions of any two equal items of input list. Say, in an input list, there are two equal

item sat positions *i* and *j* where *i< j,* then the final position of these items in the sorted list should also be k and *l* respectively, such that k<*l.* That is there should not be any swapping among these equal items and should not interchange their position with each other.

A sorting algorithm needs extra memory space to store elements during the swapping process. For small set of items in a list, this constraint is not observable but, for an input list of large elements the required storage space is considerable large. An algorithm is said to be *in-place* if the required extra memory is not markable.

Searching

Searching is finding an element, referred as *search key*, in a given set of items (may have the redundant value). Searching is one of the most important and frequently performed operation on any dataset/database.

Searching is one of the most favourite areas of researches in the field of algorithm analysis. No single searching performs optimally to all situations. Some algorithms are faster but consume more memory; some are very fast but only with specific input set; and so on.

While designing an algorithm for searching problem, it is highly influenced by the nature of underlying data. The data, static in nature, has to be addressed differently than the dynamic one in nature with addition or deletion from the data set of an item.

String Processing

Exponential increase in the textual data due to the various applications over social media and blogs, string-handling algorithms become a current area of research. Another reason for blooming strings rather text processing is the kind of data available. Now day's the business paradigms are totally changed from offline to online. According to Grant Thornton, e-commerce in India is expected to be worth US$ 188 billion by 2025.Most of the text data is used to predict the interest of people involving direct or indirect monetary benefits for commercial organizations specially e-commerce sectors. One of the most widely used search engine (Google) is also based on string processing.

Sting matching is one of the string processing problems.

Graph Problems

It is always favourable for researchers to map a computational problem to a graph problem. Many computational problems can be solved using graph. Most of the computer network problems can be solved using graph algorithms efficiently. Problems like: visiting all the nodes of a graph (broadcasting in network), routing in networks (finding the minimum cost path, i.e. the shortest

path, path with minimum delay etc. can be solved efficiently with graph algorithms.

At the same time some of the graph problems are computationally not easy, like the travelling salesman and the graph-colouring problems. The *Travelling Salesman Problem (TSP)* is used to cover *n* cities by taking the shortest path and not visiting any of the city more than once. The *graph-colouring problem* seeks to colour all the vertices of a graph with minimum number colours such that, no two adjacent vertices having the same colour. While solving TSP cities can be considered as the vertices of the graph. Event scheduling could be one of the problems which can be solved using graph colouring algorithm. Considering events to be represented by the vertices, there exists an edge between two events only if the corresponding events cannot be scheduled at the same time.

Combinatorial Problems

These types of problems have a combination of solutions i.e. more than one solution are possible. The aim of the combinatorial problems is to find permutations, combinations, or subsets, satisfying the given conditions. The travelling salesman problem, independent set and the graph-coloring problems can be categorized as examples of *combinatorial problems.* From both theoretical as well as practical point of view, the combinatorial problems are considered to be one of the most difficult problems in computing. Due to the combinatorial type of solutions, it becomes very difficult to handle the problems with big size inputs sets. The number of combinatorial objects (the output solution) grows rapidly with the problem's size.

Geometric Problems

Some of the applications of *Geometric algorithms* are computer graphics, robotics and tomography. These algorithms are based upon geometric objects such as points, lines, and polygons. The geometry procedures are developed to solve various geometric problems, like construction shapes of geometric objects, triangles, circles, etc., using ruler and compass.

Following are widely known classic problems of computational geometry:

1. The closest-pair problem
2. The convex-hull problem

The *closest-pair problem* is to find the closest pair out of a given set of points in the plane.

In the *convex-hull problem,* the smallest convex polygon is to be constructed so that it includes all the points of a given set.

Numerical Problems

Problems of numerical computing nature are simultaneous linear equations (linear algebra), differential equations, definite integration, and statistics. Most of the numerical problems could be solved approximately.

The biggest drawback of numerical algorithms is the accumulation of errors over the multiple iterations, due to rounding off the approximated result at each iteration.

1.7 PROBLEM SOLVING TECHNIQUES

Divide and Conquer Approach

This is one of the popular approaches in which a problem is divided into smaller subproblems. These subproblems are further divided into smaller subproblems until they can no longer be divided. It is a top down approach in which the algorithm *logically* progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.

An algorithm, following divide & conquer technique, involves following steps:

Step 1. Divide the problem (top level) into a set of sub-problems (lower level).

Step 2. Solve every sub-problem individually by recursive approach.

Step 3. Merge the solution of the sub-problems into a complete solution of the problem.

Following are the examples of the problems that can efficiently be solved using divide and conquer approach.

- Binary Search.
- Quick Sort.
- Merge Sort.
- Strassen's Matrix Multiplication.
- Closest Pair of Points.

Greedy Technique

Using Greedy approach, optimization problems are solved efficiently. In an optimization problem, the given set of input values are either to be maximized or minimized (called as objective), subject to some constraints or conditions.

- Greedy algorithm always picks the best choice (greedy approach) out of many at a particular moment to optimize a given objective.

- The greedy method chooses the local optimum at each step and this decision may result in overall non-optimum or optimum solution.
- The greedy approach doesn't always produce the optimal solution rather produces very nearby solution to the optimal solution.

Consequently, Greedy algorithms are often very easy to design for the optimisation problems. Following are some of the examples of the greedy approach.

- Kruskal's Minimum Spanning Tree
- Prim's Minimal Spanning Tree
- Dijkstra's shortest path
- Knapsack Problem

Dynamic Programming

Dynamic Programming approach is a bottom-up approach which involves finding solution of all sub-problems, saving these partial results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained. *Reusing* the results of sub-problems (by maintaining a table of results) is the major advantage of dynamic programming because it avoids the ***re-computations*** **(**computing results twice or more**)** of the same problem. Thus Dynamic programming approach takes much less time than naïve or straightforward methods.

The working style of dynamic programming is similar to divide and conquer approach. Both solve a problem by breaking it down into a several sub problems that can be solved recursively. The drawback of divide and conquer method is the calling of a recursive function with same output/result repeatedly which is overcome in dynamic programming by maintaining a table to store the results. It is dynamically decided whether to call a function or retrieve values from the table, that's why the word dynamic is used in it. The Dynamic programming approach is faster than the divide and conquer method as the redundancy of calling functions with same result are omitted in it.0-1 Knapsack and subset-sum problem are the examples of dynamic programming.

Branch and Bound

Branch and bound algorithm efficiently solves the discrete and combinatorial optimization problems. In branch-and-bound algorithm, a rooted tree is formed with the full solution set at the root. The algorithm explores the branches of this tree, representing the subsets of the solution set. A candidate solution of a root node is considered as a branch only if it is better than the already explored solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. Branch and Bound algorithm are methods

for solving global optimization problems. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated).

Randomized Algorithms

In a randomized algorithm, a random number is selected at any stage of the solution and is used for computation of the solution, that's why it is called as randomized algorithm. In other words it can be said that algorithms that make random choices for faster solutions are known as randomized algorithms. For example, in the Quick sort algorithm, a random number can be generated and considered as a pivot. In other example, a random number can be chosen as possible divisor to factor a large number.

Backtracking Algorithm

Backtracking algorithm is like creating checkpoints while exploring new solutions. It works analogues to depth-first search. It searches all the possible solutions. During the exploration of solutions, if a solution doesn't work, it back-track to the previous place and then find the other alternatives to get the solution. If there are no more choice points the search fails.

1.8 DETERMINISTIC AND STOCHASTIC ALGORITHM

Algorithms can be categorized either deterministic or stochastic in nature. An algorithm is deterministic if the next output can be predicted/ determined from the input and the state of the program, whereas stochastic algorithms are random in nature. Problems with unpredictable result cannot be solved using deterministic approach. For example, the next output of a card shuffling program of blackjack game should not be predictable by players even if the source code of the program is visible. Pseudorandom number generator method can be compromised and is often not sufficient to ensure the true randomness. The random number generated using Pseudorandom number generator method might be precisely predicted. To avoid this problem, use of a cryptographically secure pseudo-random number generator with an unpredictable random seed to initialize the generator can be better way. A hardware random number generator is the best way for achieving real randomness.

1.10 SUMMARY

In computation, an algorithm is independent from a programming language. Algorithm is designed to understand and analyze the solution of a computational problem. In an algorithm, statements may be used to perform an action called as sequencing, or to take decision (selection) or to repeat certain actions (iterations).

Running time of an algorithm is one of the most widely used parameter to judge an algorithm. Running time is computed as the number of instructions executed. Space complexity is measurement of memory storage used during the execution. For a computational problem, there may exist multiple algorithms to solve which leads to analyze these Algorithms to get the best solution as per the requirement.

An algorithm can be analyzed in terms of time complexity and space complexity. To evaluate algorithms of a problem, time and space complexities are considered. Algorithm, taking less time to produce the desired output, is desirable. There has to be a tradeoff between these two parameters, an algorithm with less space complexity may not be desirable if it runs for a long time.

In computing, based on the nature of the problem, it can be assigned any one of the commonly known categories, namely sorting, searching, string processing, graph problems, combinatorial problems, geometric problems, numerical problems.

Similar type of problems can be solved with similar approach. Some of the commonly used problems solving techniques are Brute Force and Exhaustive search approach, Divide and Conquer approach, Greedy technique, Dynamic Programming, Branch and Bound, Randomized algorithms, and Backtracking algorithm.

Another widely known categories of Algorithms, based on the type of the inputs used, are Deterministic and Stochastic Algorithms. If the output of an algorithm can be predicted by looking at the input, such algorithms are called as deterministic in nature. While stochastic algorithms are random in nature, means the output cannot be determined from the input.

1.11 SOLUTION TO CHECK YOUR PROGRESS

Q1. What is an Algorithm? What are the important characteristics of an algorithm? Solution: An Algorithm is a set of steps to solve a problem or a set of problems. Also, an algorithm is a step by step procedure to solve logical and mathematical or computational problems. A recipe is a good example of Algorithm. To cook a dish a recipe says what to be done step by step.

Important characteristics of an algorithm are: input, output, definiteness, effectiveness and finiteness

Q2.What are the building blocks of an Algorithm?

Selection, selection, iteration, procedure and recursion

Q3. How to judge an algorithm, whether it is efficient or not?

Solution: An algorithm should be both correct and efficient. The efficiency of an Algorithm is defined in terms of the resource usage to yield a correct answer.

In general the efficiency of an Algorithm is considered in terms of the computational complexity, which is: how hard is it to execute the algorithm. The execution of an Algorithm is very much depends on the input. An Algorithm may perform differently depending on how input looks like. For example: The efficiency of Insertion sort Algorithm depends on the input array. Here, insertion sort has a linear running time (i.e., O(n)). While it is quadratic running time (i.e., O(n2)) for an array sorted in reverse order known as the worst case.

Q4. Map the Set B with corresponding problems listed in Set B.

S. N.	Set A	S.N.	Set B
1	Sorting Problem	A	Arranging a list of numbers in ascending order.
2	Geometric Problem	B	Finding an item in set items.
3	Graph Problem	C	Finding the shortest path between two nodes.
4	Numerical Problem	D	Finding Euler graph for a given graph.
5	Searching Problem	E	Finding the solutions of a given set of linear equations.
6	String Processing	F	Finding the pair of points (from a set of points) with the smallest distance between them.
7		G	Match a word in a paragraph.

Solution: The correct match is as follows:

1 – A, 2 – F, 3 – C, D, 4 – E, 5 – B, 6 – G.

Q5. When should the deterministic approach of problem solving to be avoided? Explain with an example.

Solution: An algorithm is deterministic if the next output can be predicted/ determined from the input and the state of the program. Deterministic approach of problem solving technique is not suitable for the problems with unpredictable result.

For example, the next output of a card shuffling program of blackjack game should not be predictable by players even if the source code of the program is visible.

Q6. Differentiate Dynamic programming and backtracking problem solving approach. What problems can be solved by each technique?

Solution: Dynamic programming is a technique widely used to solve optimization problem. Optimization problem is used to find the either minimum or maximum result (a single result) out of all possible outcomes. In backtracking method, a brute force approach is used, hence it is not used for optimization problem. Backtracking approach is suitable for solving problems having multiple results and out of which, all or some of them are acceptable.

Following problems can be solved using backtracking approach:

- Eight queen puzzle
- Map coloring
- Sudoku

Following problems can be solved using dynamic programming approach:

Ans

- All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford
- 0/1 knapsack problem
- Chain matrix multiplication
- Traveling salesman problem

Q7 What problems can be solved through greedy technique?

Ans

- Fractional knapsack problem
- Minimum cost spanning tree
- Single source shortest path algorithm

Q9 What are the common running times for the algorithms?

Ans. Constant time, linear time, logarithmic time, polynomial time, exponential time and factorial time

Q10Define independent set problem.

Ans. Independent set problem can be defined as a set of nodes which are not joined by any edge. One way to formulate this problem is that given a constant k and a graph G having n nodes (vertices) find out an independent set of a size k.

1.12 FURTHER READINGS

1. Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. Algorithm Design, Jon Kleinberg and Eva Tardos, Pearson

UNIT 2 ASYMPTOTIC BOUNDS

Structure

2.0 INTRODUCTION

In the last unit, we have discussed about algorithms and its basic properties. We also discussed about deterministic and stochastic algorithms. In this unit, we will discuss the process to compute complexities of different algorithms, useful mathematical functions and notations, principle of mathematical induction, and some well known asymptotic functions. Algorithmic complexity is an important area in computer science. If we know complexities of different algorithms then we can easily answer the following questions-

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

The above-mentioned criterions are used as the basis of the comparison among different algorithms. With the help of algorithmic complexity, programmers improve the quality of their code using relevant data structures. To measure the efficiency of a code/ algorithm, asymptotic notations are normally used. Asymptotic notations are the mathematical notations that estimate the time or space complexity of an algorithm or program as function of the input size. For example, the best-case running time of a function that sorts a list of numbers using bubble sort will be linear i.e., O(n). On the contrary, the worst- case running time will be $O(n^2)$. So, we can say that the bubble sort takes T(n) time, where, $T(n)=O(n^2)$. The asymptotic behavior of a function f(n) indicates the the growth of f(n) as n gets very large. The small values of n are generally ignored as we are interested to know how slow the program or algorithm will be on large input. The slower asymptotic growth rate, the better the algorithm performance. As per this measurement, a linear algorithm (i.e., $f(n)=d*n+k$) is always asymptotically better than a quadratic one (e.g., $f(n)=c*n^2+q$) for any positive value of

c, k, d, and q. To understand concepts of asymptotic notations, you will be given a idea of lower bound, upper bound, and an average bound. Mathematical induction plays an important role in computing the algorithms' complexity. Using the mathematical induction, problem is converted in the form mathematical expression which is solved to find the time complexity of algorithm. Further to rank algorithms in increasing or decreasing order asymptotic notations such as big oh, big omega, and big theta are used.

2.1 OBJECTIVES

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

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

2.2 SOME USEFUL MATHEMATICAL FUNCTIONS & NOTATIONS

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

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

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

Notation: If a1, a2…an are n real variables/numbers then

2.2.1 Summation & Product

Summation:

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

Sum of sequences

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

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

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

Product

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

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

2.2.2 Function:

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

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

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

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

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

Next, we state a useful result, without proof.

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

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

2.2.3 Logarithms

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

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

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

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

(i) $\log_a(bc) = \log_a b + \log_a c$

(ii) $\log_a(b^n) = n\log_a b$

(iii) $\log_b a = \log_a b$

(iv) $\log_a(1/b) = -\log_b a$

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

(vi) ${}_a\log_b c = {}_c\log_b a$

Modular Arithmetic/Mod Function

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

Definition

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

b mod n=rwhere0 ≤ r <n and b = k * n +r

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

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

b mod n = 42 mod 11 = 9.

If b = −42andn = 11then

b mod n = −42 mod 11 = 2 (∵−42 = (−4) × 11 + 2)

2.3 SOME MATHEMATICAL EXPECTATION

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

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

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

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

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

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

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

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

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

Thus

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

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

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

Mathematical Expectation

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

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

2.4 THE PRINCIPLE OF INDUCTION

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

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

Therefore the method consists of the following three major steps:

1. **Induction base-** In this stage we verify/establish the correctness of the initial

value (base case). It is the proof that the statement is true for n = 1 or some other starting value.

2. Induction Hypothesis- it is the assumption that the statement is true for any value of n where n ≥1
3. Induction Step- In this stage we make a proof that if the statement is true for n , it must be true for n+1

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

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

Proof: (Through Induction Method)

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

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

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

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

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

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

Consider

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

Let us rewrite the above equation in the following way:

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

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

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

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

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

Check your progress-1

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

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

Questioin2: Prove that for all nonnegative integers n,

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

2.5 EFFICIENCY OF AN ALGORITHM

In order to understand the complexity/efficiency of an algorithm, it is very important to understand the notion of the **size of an instance** of the problem under consideration and the role of size in determining complexity of the solution. For example finding the product of two 2 × 2 matrices will take much less time than the time taken by the same algorithm for multiplying say two 100 × 100 matrices. This explains intuitively *the notion of the size of an instance of a problem* and also the role of size in determining the (*time*) complexity of an algorithm. **If the *size* (to be later considered**

formally) of general *instance is n* then time complexity of the algorithm solving the *problem* (not just the instance) under consideration is *some function of n.*

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

(i) In sorting and searching problems, the *number of elements*, which are to be sorted or are considered for searching, is taken *as the size of the instance* of the problem of sorting/searching.
(ii) In the case of solving polynomial equations or while dealing with the algebra of polynomials, the *degrees* of polynomial instances, may be taken as the sizes of the corresponding instances.
To measure the efficiency of an algorithm, we will consider the theoretical approach and follow the following steps:

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

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

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

2.6 Asymptotic Analysis and Notations

Asymptotic analysis" is a more formal method for analyzing algorithmic efficiency. It is a mathematical tool to analyze the time and space complexity of an algorithm as a function of input size. For example, when analyzing the time complexity of any sorting algorithm such as Bubble sort, Insertion sort and Selection sort in the worst case scenario , we will be concerned with how long it takes as a function of the length of the input list. For example, we say the time complexity of standard sorting

algorithms in the worst case takes T(n) = n^2 where n is a size of the list . In contrast, Merge sort takes time $T(n) = n*log_2(n)$.

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

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

$T_1(n) = 1000\ n^2$

$T_2(n) = 5n^4$

Despite the fact $T_1(n) \geq T_2(n)$ for $n \leq 14$, we would still prefer the solution as $T_1(n)$ as the time complexity because

$T_1(n) \leq T_2(n)$ for all $n \geq 15$

Some common orders of growth seen often in complexity analysis are

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

2.6.1 Worst Case and Average Case Analysis

Consider a linear search algorithm. The worst case of the algorithm is when the element to be searched for is either not in the list or located at the end of the list. In this case the algorithm runs for the longest possible time. It will search the entire list. If an algorithm runs in time *T(n)*, we mean that *T(n)* is an upper bound on the running time that holds for all inputs of size *n*. This is called *worst-case analysis*.

A popular alternative to worst-case analysis is *average-case analysis* which provides average amount of time to solve a problem. Here we try to calculate the expected time spent on a randomly chosen input. This kind of analysis is generally more difficult compared to worst case analysis. Because it involves probabilistic arguments and often requires assumptions about the distribution of inputs that may not be very easy to justify. But sometimes it can be more useful compared to the worst-case analysis of an algorithm. A Quick sort algorithm, whose worst-case running time on an input

sequence of length n is proportional to n^2 but whose average running time is proportional to n log n.

2.6.2 Drawbacks of Asymptotic Analysis

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

2.6.3 Asymptotic Notations

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

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

2.6.4 Big-O notation: Upper Bounds

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

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

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

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

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

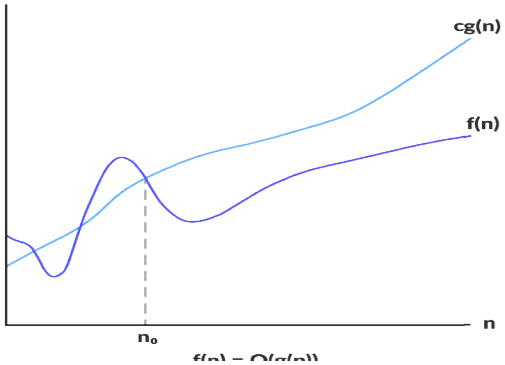


Figure f(n) = O(g(n))

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

Consider the following examples

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

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

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

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

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

$= O(n^2)$

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

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

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

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

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

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

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

First select c and n_0 so that:

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

Now both sides are divided by n^2

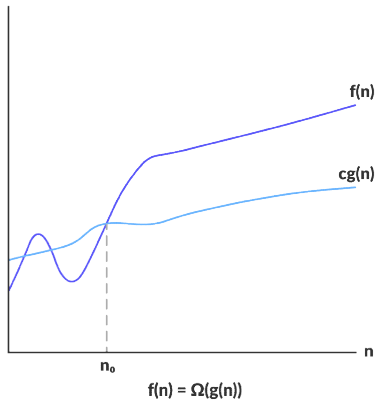
$n<=c$ for all $n>=n_0$

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

2.6.2 Big-Omega(Ω) Notation

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

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



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

Example 1

Example1.1: For the function defined by

$f(n) = 2n^3 + 3n^2 + 1$ *and*

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

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

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

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

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

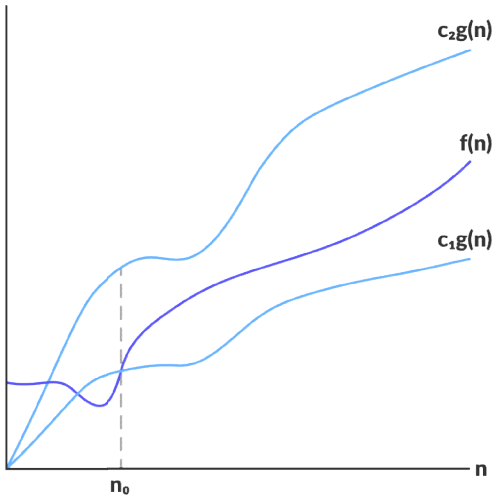
clearly this equation (1) *is satisfied for* $C = 1$ *and for all* $n \geq 1$

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

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

2.6.3 Θ (Theta) notation: Tight Bounds

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



f(n) = Θ(g(n))

Now let us define the theta notation: for a given function g(n) and constants C_1,C_2 and n_0 where $n_0>0$, $C_1>0$, and $C_2>0$, $\Theta(g(n))$ can be denoted as a set of functions such that the following condition is satisfied:

$0 <= C_1g(n) <= f(n) <= C_2g(n)$ for all $n >= n_0$

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

The following theorem which relates the three functions O, Ω,Θ.

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

if f(n) is Θ(g(n)) this means that the growth of f(n) is asymptotically at the same rate as g(n) or we can say the growth f(n) is not asymptotically slower or faster than g(n) no matter what value of n is provided

2.6.4 Some Useful Theorems for O, Ω, Θ

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

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

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

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

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

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

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

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

Then $f(n) \leq cn^m \Longrightarrow f(n) = O(n^m)$.

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

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

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

Since $f(n) \geq cn^m\ for\ all\ n \geq 1 \Longrightarrow f(n) = \Omega(n^m)$

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

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

Proof: From Theorem 1 and Theorem 2,

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

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

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

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

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

Solution:

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

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

Check your progress-2

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

a) $4n^2 + 7n + 12 = O(n^2)$
b) $logn + \log(logn) = O(logn)$
c) $3n^2 + 7n - 5 = \Theta(n^2)$
d) $2^{n+1} = O(2^n)$
e) $2^{2n} = O(2^n)$
f) $f(n) = O\ (g(n)) implies\ g(n) = O(f(n))$
g) $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$
h) $f(n) = O(g(n)) implies\ 2^{f(n)} = O(2^{g(n)})$
i) $f(n) + g(n) = \Theta(\min(f(n), g(n))$

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

k) $f(n) + g(n) = O(n^2)\ where$
$f(n) = 2n^2 - 3n + 5\ and\ g(n) = nlogn + 10$

(

2.7 SUMMARY

In computer science, before solving any problem algorithm is designed. An algorithm is a definite, step-by-step procedure for performing some task. In general, an algorithm takes some sort of input and produces some sort of output. Usually, it is required that an algorithm is guaranteed to terminate after a finite amount of time and returns the correct output. Algorithm must be efficient and easy to understand. To analyze the algorithms efficiency, asymptotic notations are generally used. There are three popular asymptotic notations namely, big O, big Ω, and big Θ. Big omega notation is mostly used as it measure the upper bound of complexity. On the contrary, big omega measures the lower bound of time complexity and big theta is used to measure the average time of any algorithm.

2.8 SOLUTION TO CHECK YOUR PROGRESS

Check Your Progress 1

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

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

Proof: (Base Step):
Consider n=1, then

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

Hence for n=1 it is true.
Induction Hypothesis: Assume the above statement is true for 'n' i.e.

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

Induction Step:
Now we need to show that

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

To that end,

$$\begin{aligned} 1^2 + 2^2 + \cdots + (n+1)^2 &= 1^2 + 2^2 + \cdots + n^2 + (n+1)^2 \\ &= \frac{n(n+1)(2n+1)}{6} + (n+1)^2 \\ &= \frac{n(n+1)(2n+1) + 6(n+1)^2}{6} \\ &= \frac{(n+1)(2n^2 + n + 6n + 6)}{6} \end{aligned}$$

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

Questioin2: Prove that for all nonnegative integers n,

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

We show, for all nonnegative integers n, that

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

In summation notation, this equality can be defined as

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

Induction base: For n = 0,

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

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

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

Induction step: We need to show that

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

To that end,

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

Check Your Progress 2

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

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

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

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

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

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

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

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

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

$\Rightarrow$ 1+log(1) ≤ 2

$\Rightarrow$ 1+0≤ 2

$\Rightarrow$ Satisfied for c = 2 and $n_0 = 2$

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

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

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

To show this, we have to show:

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

(i) L.H.S inequality

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

This is satisfied for $C_1 = 1 \text{and} n \geq 2$

(ii) R.H.S inequality

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

This is satisfied for $C_1 = 1\ and\ n \geq 1$

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

$C_1 = 1, C_2 = 10\ and\ n \geq 2$

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

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

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

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

$\Rightarrow \quad 4^n \leq 2.2^n \ldots\ldots(1)$

No value of C and n_0Satisfied this in equality (1)

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

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

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

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

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

To prove this, we have to show that

$C_1.(f(n) + g(n) \leq \max\{f(n), g(n)\}$

$\leq C_2(f(n)+g(n) \ldots\ldots\ldots\ldots(*)$

1) L.H.S inequality:

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

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

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

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

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

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

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

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

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

2) R.H.S inequality

$\max\{f(n), g(n)\} \leq C_1.\ (f(n) + g(n)$ ……(2)

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

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

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

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

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

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

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

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

we can prove this by taking a counter Example ;

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

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

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

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

We can prove this by taking counter example.

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

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

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

There is no positive integer for C and n_0

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

k) $f(n) + g(n) = O(n^2)\ where$

$f(n) = 2n^2 - 3n + 5\ and\ g(n) = nlogn + 10$

$f(n) + g(n) = 3n^2 - n + 4 + nlogn + 5 = h(n)$

By O = *notation* $h(n) \leq cn^2$;

This is true for c = 4 and $n_0 \geq 4$

UNIT 3 COMPLEXITY ANALYSIS OF SIMPLE ALGORITHMS

Structure **Page Nos.**

3.0 INTRODUCTION

Computational complexity describes the amount of processing time required by an algorithm to give the desired result. Generally we consider the worst-case time complexity (big O notation) which is the maximum amount of time required to execute an algorithm for inputs of a given size. Whereas average-case complexity, which is the average of the time taken on inputs of a given size is less common. In the previous unit we introduced a concept of efficiency of an algorithm and discussed three asymptotic notations which are formal methods for analyzing algorithm efficiency in terms of time and space complexities. The complexity analysis of algorithm helps to understand the behavior of the algorithm and compare it with other algorithms for a large input size. The structure of the unit is as follows: section 3.3makessimple statements, a brief review of asymptotic notations followed by analysis of simple constructs such as looping statement, conditional statement, and etc. .In the subsequent sections we describe general rules for analysis of algorithms and illustration with several examples.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- Algorithm to add n cube numbers
- An algorithm to evaluate polynomial by Horner's rule
- Analysis of Matrix Multiplication algorithm
- Define big oh, big omega and big theta notation
- Exponent evaluation in logarithmic complexity
- Linear search and its complexity analysis
- Basic sorting algorithm and their analysis

3.2 A BRIEF REVIEW OF ASYMPTOTIC NOTATIONS

The complexity analysis of algorithm is required to measure the time and space required to run an algorithm. In this unit we focus on only the time required to execute an algorithm. Let us quickly review some asymptotic notations (Please refer to the previous unit for detailed discussion)

The central idea of these notations is to compare the relative rate of growth of functions.

Assume T(n) and f(n) are two functions

(i) $T(n) = O(f(n))$ if there are two positive constants C and n_0 such that $T(n) \leq Cf(n)$ where $n \geq n_0$

(ii) $T(n) = \Omega(f(n))$ if there are two positive constants C and n_0 such that $T(n) \geq C\Omega$ f(n) where $n \geq n_0$

(iii) $T(n) = \theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

I Assume $T(n) = 1000n and f(n) = n^3$. A function 1000 n is larger than n^3 for small value of n, but n^3 will grow at faster rate if the value n become large. Therefore n^3 is a larger function. The definition of $T(n) = O(f(n))$ says that T(n) will grow slower or equal to C. f(n) after the point where $n \geq n_0$.

The second definition, $T(n) = \Omega(f(n))$ says that the growth rate of T(n) is faster than or equal to (≥) f(n).

3.3 ANALYSIS OF SIMPLE CONSTRUCTS OR CONSTANT TIME

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) 'if (i) statements are simple statement like assignment, increment or decrement operation and declaration statement and (ii) there is no recursion, loops or call to any other function.

Ex. $int\ x;$

$x = x + 5$

$x = x - 5$

2) O(n): This is running time of a single looping statement which includes comparing time, increment or decrement by some constant value looping statement.

```
// Here c is a positive integer constant
  for (i = 1; i <= n; i += c) {
     // simple statement(s)
```

```
}

for (int i = n; i> 0; i -= c) {
    // simple statement(s)
}
```

3) O(n^c): This is a running time of nested loops. Time complexity of nested loops is equal to the number of times the innermost statements is executed. For example, the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i<=n; i += c) {
    for (int j = 1; j <=n; j += c) {
      // some simple statements
    }
}
```

Most of the simple sorting algorithms

have $O(n^2)$ time complexity in the worst case.

4) O(logn) If the loop index in any code fragment is divided or multiplied by a constant value, the time complexity of the code fragment is O(logn)

```
for (int i = 1; i<=n; i *= c ) {
    // some simple statements
}
for (int i = n; i> 0; i /= c) {
   // simple statements
}
```

5) Time complexities of consecutive loops

If the code fragment is having more than one loop, time complexity of the fragment is sum of time complexities of the individual loops.

```
for (int i = 1; i<=m; i ++ c) {
// simple statements taking θ(1)
}
for (int f = 1; i<=n; if += X) {
    // simple statements of θ(1)
}
```

Time complexity of above fragment is O(m) + O(n) which is O(m+n)

If one looping statement is consecutive if-else statement

6) Consecutive if else statements
The running time of the if-else statement is just running time of the testing the condition plus the larger of the running of times statement1 or statement2

code fragment of
if – else is

```
if (condition)
    statement 1
else
    statement 2
```

3.4 ANALYSIS OF SIMPLE ALGORITHMS

In this section we will illustration analysis of simple algorithms to simplify the complexity analysis we will apply the following general rules.

- By default it is big oh running time.
- No consideration of low-order terms.
- No consideration of constant value.

3.4.1 A Summation Algorithm

The following is a simple program to calculate $\sum_{i=1}^{n} i^3$

```
int sum of n cube (int n)
{
1. int i, tempresult;
2. tempresult =0;
3. for (i=1 ; I <=n;  i++)
4. tempresult = tempresult + i * i * i
5. return tempresult;
}
```

Line#1- 2 units of time required for declaration.

Line# 2- 1 unit of time required for assignment operation.

Line# 3- The **for** loop has several unit costs: initializing i, cost for testing i<=n

(n+1 unit cost) and cost of incrementing i(1 unit of cost) Total cost is 2n +2

Line# 4- 2units of time for multiplication, 1 unit for addition and one unit of time for assignment operation in one cycle. Therefore the total cost of this line is 4n

Line# 5- It will take 1 unit of time. Overall cost will be = 6n+6 which is written as O(n).

3.4.2 Polynomial Evaluation

A polynomial is an expression that contains more than two terms. A term comprises of a coefficient and an exponent.
Example: $P(x) = 15x^4+7x^2+9x+7$ $P(x)=14x^4+17x^3-12x^2+13x+16$

A polynomial may be represented in form of array or structure. A structure representation of a polynomial contains two parts – (i) coefficient and (ii) the corresponding exponent. The following is the structure definition of a polynomial:

```
Struct polynomial{

        int coefficient;

        int exponent;

};
```

How to evaluate the polynomial? It can be evaluated through brute force method and Horner's method. Let us try to understand through an example

Consider the polynomial

Suppose that exponentiation is implemented through multiplications. The processes of evaluation through both the methods are shown below:

Brute force method:

$P(x)=15*x*x*x*x+17*x*x*x-12*x*x+13*x+16$

Horner's method:

$P(x)=(((15*x+17)*x-12)*x+13)*x+16$
Please observe the basic operations are: multiplication, addition and subtraction. Since the number of additions and subtractions are the same in both the solutions, we will consider the number of multiplications only in worst case analysis of both the methods.
[The general form of a polynomial of degree n, and express our result in terms of n. We'll look at the worst case (maximum number of multiplications) to get an upper bound on the work]

Now consider the general form of a polynomial of degree n is

$P(x)=a_nx^n+a_{n-1}x^{n-1}+.....+a_1x^1+a_0x^0$

where $a_0,a_1,.....a_{n-1},a_n$ are coefficients and $x^0, X^1, \ldots X^{n-1}, X^n$ are related exponents and we have to evaluate polynomial at a specific value for x.

(i) Analysis of Brute Force Method

A brute force approach to evaluate a polynomial is to evaluate all terms one by one. First calculate x^n, multiply the value with the related coefficient a_n ,repeat the same steps for other terms and return the sum

$P(x)= a_n*x*x* \ldots*x*x + a_{n-1}*x*x* \ldots *x*x + a_{n-2}*x*x* \ldots*x*x + \cdots$

$+a_2*x*x*+a_1*x+a_0$

In the first term, it will take n multiplications, in the second term n-1 multiplications, in the third term it takes n-2multiplications….. In the last two terms: $a_2*x*x*$and a_1*xit takes 2 multiplications and 1 multiplicationaccordingly.

Number of multiplications needed in the worst case is

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

$=n(n+1)/2= O(n^2)$

(ii) Analysis of Horner's Method

$P(x)=(\ldots(((a_n*x+a_{n-1})*x+a_{n-2})*x+\ldots+a_2)*x+a_1)*x+a_0$

In the first term it takes one multiplication, in the second term one multiplication, in the third term it takes one multiplication …. . Similarly in all other terms it will take one multiplication.

Analysis of Horner's Method

Number of multiplications needed in the worst case is:

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

$T(n)=n$

Consider only the first term of a polynomial of degree ***n***: $a_n x^n$ to appreciate the efficiency of Horner's rule .Just computing this single term by the brute-force algorithm would require ***n*** multiplications, whereas Horner's rule requires only one multiplication in every term.

(iii) Pseudo code for polynomial evaluation using Horner method, Horner(a,n,x)

//In this poly is an array of n elements which are coefficient of polynomial of degree n

1. Assign value of poly p[n]= coefficient of nth term in the polynomial
2. set i=n-1
4. compute p = p * x +
poly[i];
5.i=i-1
6. if i is greater than or equal to 0 Go to step4.

7. final polynomial value at x is p.

StepII.AlgorithmtoevaluatepolynomialatagivenpointxusingHorner'srule:

Input: An array A[0..n] of coefficient of a polynomial of degree n and a point x

Output: The value of polynomial at given point x

```
Evaluate_Horner (a,n,x)
{
p = A[n];
for (i = n-1; i≤0;i--)
        p = p * x + A[i];
return p;
}
```

For Example: $p(x)=3x^2+5x+6$ using Horner's rule can be simplified as

follows

At x=3,
p(x) = (3x+5)x+6
p(2)=(9+5).3+6
= (14).3+6
=42+6
=48

Complexity Analysis

Polynomial of degree n using Horner's rule is evaluated as below:
Initial assignment, p = a[n]
After the first iteration p = $xa_n + a_{n-1}$
After the second iteration, p = $x(xa_n + a_{n-1}) + a_{n-2}$

$= x^2a_n + x\,a_{n-1} + a_{n-2}$

Every subsequent iteration uses the result of previous iteration i.e next iteration multiplies the previous value of p then adds the next coefficient, i.e.

$p = x(x^2a_n + x\,a_{n-1} + a_{n-2}) + a_{n-3}$

$= x^3a_n + x^2a_{n-1} + x\,a_{n-2} + a_{n-3}$etc.

Thus, after n iterations, $p = x^n a_n + x^{n-1} a_{n-1} + \ldots + a_0$, which is the required correct value.

In above function

First step is one initial assignment that takes constant time i.e O(1).

For loop in the algorithm runs for n iterations, where each iteration cost O(1) as it includes one multiplication, one addition and one assignment which takes constant time.

Hence total time complexity of the algorithm will be O(n) for a polynomial of degree n.

☞Check Your Progress 1

1. Define polynomial. Write the expression of polynomial of degree.

……

……

……

2 Evaluate $p(x)= 3x^4+2x^3-5x+7$ at x=2 using Horner's rule. Show step wise iterations.

……

……

……

3. Write basic algorithm to evaluate a polynomial and find its complexity. Also compare its complexity with complexity of Horner's algorithm.

……

……

……

MATRIX (N X N)MULTIPLICATION

Matrix is very important tool in expressing and discussing problems which arise from real life cases. By managing the data in matrix form it will be easy to manipulate and obtain more information. One of the basic operations on matrices is multiplication.

In this section matrix multiplication problem is explained in two steps as we have discussed GCD and Horner's Rule in previous section. In the first step we will brief pseudo code and in the second step the algorithm for the matrix multiplication will be discussed. This algorithm can be easily coded into any programming language.
Further explanation of the algorithm is supported through an example.

Let us define problem of matrix multiplication formally, then we will discuss how to multiply two square matrix of order n x n and find its time complexity. Multiply two matrices A and B of order n*n each and store the result in matrix C of order n*n.

A square matrix of order n*n is an arrangement of set of elements in n rows and n columns.

Let us take an example of a matrix of order 3 x 3 which is represented as

$$A= \begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}_{3 \times 3}$$

This matrix A has 3 rows and 3 columns.

Step I : Pseudo code: For Matrix multiplication problem where we will multiply two matrices A and B of order 3x3 each and store the result in matrix C of order 3x3.

1. Multiply first row first element of first matrix with first column first element of second matrix.
2. Similarly perform this multiplication for first row of first matrix and first column of second matrix. Now take the sum of these values.
3. The sum obtained will be first element of product matrix C
4. Similarly Compute all remaining element of product matrix

C.

i.e $c_{11}= a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31}$

C= A x B

Step II : Algorithm for multiplying two square matrix of order n x n and find the product matrix of order n x n

Input: Two n x n matrices A and B
Output: One n x n matrix C = A x B

```
Matrix_Multiply(A,B,C,n)
{

for i = 0 to n-1 //outermost loop
      for j = 0 to n-1
      {
      C[i][j]=0            //assignment statement
                  for k = 0 to n-1 // inner most loop
                  C[i][j] = C[i][j] + A[i][k] * B[k][j]
      }
}
```

For Example matrix A (3 x 3) , B(3 x3)

$$A = \begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{matrix}$$

$$B = \begin{matrix} 1 & 1 & 1 \\ 2 & 3 & 2 \\ 3 & 2 & 1 \end{matrix}$$

To compute product matrix C= A x B

$$\begin{matrix} c_{11} & c_{12} & c_{13} \\ {}_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{matrix} = \begin{matrix} 1x1+2x2+3x3 & 1x1+2x3+3x21x1+2x2+3x1 \\ 2x1+3x2+4x3 & 2x1+3x3+4x22x1+3x2+4x1 \\ 4x1+5x2+6x3 & 4x1+5x3+6x24x1+5x2+6x1 \end{matrix}$$

$$= \begin{matrix} 14 & 13 & 8 \\ 20 & 19 & 12 \end{matrix}$$

32 31 20

Complexity Analysis

First step is, for loop that will be executed n number of times i.e. it will take O(n) time. The second nested for loop will also run for n number of time and will take O(n) time.

Assignment statement inside second for loop will take constant time i.e. O(1) as it includes only one assignment.

The third for loop i.e. innermost nested loop will also run for n number of times and will take O(n) time . Assignment statement inside third for loop will cost O(1) as it includes one multiplication, one addition and one assignment which takes constant time.

Hence, total time complexity of the algorithm will be $O(n^3)$ for matrix multiplication of order n*n.

☞Check Your Progress 2

1. Write a program in 'C' to find multiplication of two matrices A[3x3] and B[3x3].

……

……

……

3.4.3 EXPONENTEVALUATION

Exponent evaluation is the most important operation. It has applications in cryptography and encryption methods, The **exponent** tells us how many times to multiply the base by itself. Raising a to the power of n is expressed as multiplication by a done n-1 times:$a^n=a\cdot a\cdot\ldots\cdot a$.However, this method is not practical for large a or n.Therefore we will apply the binary exponentiation method .The idea of binary exponentiations, to split the exponent using the binary representation and then do multiplication work of. Let's write n is base 2, for example:

$4^{11}=4^{10112}=4^8\cdot4^2\cdot4^1$

Since the number n has exactly$\lfloor\log_2 n\rfloor+1$digits in base2,we only need to perform $O(\log n)$ multiplications, if we know the powers $a^1, a^2, a^4, a^8, \ldots, a^{\lfloor\log n\rfloor}$.

The following example illustrates the intermediate steps in binary exponentiation. Every subsequent multiplication is just the square of the previous multiplication

$4^1=4$

$4^2=(4^1)^2=4^2=16$

$4^4=(4^2)^2=16^2=256$

$4^8=(4^4)^2=256^2=65{,}536$

Therefore the final answer for 4^{11}, we only need to multiply three of them (skipping

4^4 because the corresponding bit in n is set to zero):$4^{11}=65,536\cdot16\cdot4=4,194,304$.

The time complexity of this algorithm is $(\log n)$: to compute $logn$ power of a and then to do almost $\log n$ multiplications to get the final result from them.

Computing x^n at some point x = a i.e a^n tends to brute force multiplication of a by itself n-1 times. So. To reduce the number of multiplication binary exponentiation methods to compute x^n will be discussed. Processing of binary string for exponent n to compute x^n can be done by following methods:

- left to right binary exponentiation
- right to left binary exponentiation

Left to right binary exponentiation

In this method exponent n is represented in binary string. This will be processed from left to right for exponent computation x^n at x=a i.ea^n .First we will discuss its pseudo code followed by algorithm.

Step I : Pseudo code to compute a^n by left to right binary exponentiation method

// An array A of size s with binary string equal to exponent n, where s is length of binary string n

1. Set result=a
2. Set i=s-2
3. compute result = result *result
4. if A[i] = 1 then compute result = result *a
5. i=i-1 and if i is less than equal to 0 then go to step4.
6. return computed value as result.

Step II : Algorithm to compute a^n by left to right binary exponentiation method is as follows:

Input: a^n and binary string of length s for exponent n as an array A[s]

Output: Final value of a^n.

1. result=a
2. for i=s-2 to0
3. result = result *result
4. if A[i]= 1then
5. result= result *a
6. return result (i.e a^n)

Let us take an example to illustrate the above algorithm to compute a^{17}
In this exponent n=17 which is equivalent to binary string 10001
Step by step illustration of the left to right binary exponentiation algorithm for a^{17} :
s=5

result=a

Iteration 1:

i=3
result=a *a= a^2

$A[3] \neq 1$

Iteration 2:

i=2
result= $a^2 * a^2 = a^4$
$A[2] \neq 1$

Iteration 3:

i=1
result= $a^4 * a^4 = a^8$
$A[1] \neq 1$

Iteration 4:

i=0
result= $a^8 * a^8 = a^{16}$
$A[0] = 1$
result = $a^{16} * a = a^{17}$

return a^{17}

In this example total number of multiplication is 5 instead of 16 multiplications in brute force algorithm i.e n-1

Complexity analysis: This algorithm performs either one multiplication or two multiplications in each iteration of a for loop in line no. 2 of the algorithm.

Hence

Total number of multiplications in the algorithm for computing a^n will be in the range of $s-1 \leq f(n) \leq 2(s-1)$ where s is length of the binary string equivalent to exponent n and f is function that represent number of multiplication in terms of exponent n. So complexity of the algorithm will be $O(\log_2 n)$ As n can be representation in binary by using maximum of s bits i.e $n=2^s$ which further implies $s= O(\log_2 n)$

Right to left binary exponentiation

In right to left binary exponentiation to compute a^n , processing of bits will start from least significant bit to most significant bit.

Step I : Pseudo code to compute a^n by right to left binary exponentiation method

// An array A of size s with binary string equal to exponent n, where s is length of binary string n

1. Set x =a
2. if A[0]= 1 then set result=a
3. else set result=1
4. Initialize i=1
5. compute x = x *x
6. if A[i] = 1 then compute result = result *x
7. Increment i by 1 as i=i+1 and if i is less than equal to s-1 then go to step4.
8. return computed value as result.

Step II : Algorithm to compute a^n by right to left binary exponentiation method algorithm is as follows:

Input: a^n and binary string of length s for exponent n as an array

A[s] Output: Final value of a^n.

```
1.    x=a
2.    if A[0]=1then
3.          result =a
4.    else
5.          result=1
6.    for i= 1 tos-1
7.         x= x * x
8.         if A[i]=1
9.               result= result *x
10.   return result (i.e a^n)
```

Let us take an example to illustrate the above algorithm to compute a^{17}In this exponent n=17 which is equivalent to binary string 10001

Step by step illustration of the right to left binary exponentiation algorithm for a^{17} : s=5, the length of binary string of 1's and 0's for exponent n

Since A[0] =1 , result=a

Iteration 1

i=1
x=a *a=
a^2
A[1] ≠
1

Iteration 2

i=2
x= a^2 * a^2 = a^4
A[2] ≠ 1

Iteration 3

i=3
x= a^4 * a^4 = a^8
A[3] ≠ 1

Iteration 4

i=4
x= a^8 * a^8 = a^{16}
A[4] = 1
result = result * x = a * a^{16} = a^{17}

return a^{17}

In this example total number of multiplication is 5 instead of 16 multiplications in brute force algorithm i.e n-1

Complexity analysis: This algorithm performs either one multiplication or two multiplications in each iteration of for loop as shown in line no. 6.
Hence

Total number of multiplications in the algorithm for computing a^n will be in the range of $s-1 \leq f(n) \leq 2(s-1)$ where s is length of the binary string equivalent to exponent n and f is function that represent number of multiplication in terms of exponent n. So complexity of the algorithm will be $O(\log_2 n)$ As n can be representation in binary by using maximum of s bits i.e $n=2^s$ which further implies $s= O(\log_2 n)$

From the above discussion we can conclude that the complexity for left to right binary exponentiation and right to left binary exponentiation is logarithmic in terms of exponent n.

☞Check Your Progress 3

1. Compute a^{283} using left to right and right to left binary exponentiation.

……

……

……

Linear Search

Linear search or sequential search is a very simple search algorithm which is used to search for a particular element in an array. Unlike binary search algorithm, in linear search algorithm, elements are not arranged in a particular order. In this type of search, an element is searched in an array sequentially one by one. If a match is found then that particular item is returned and the search process stops. Otherwise, the search continues till the end of an array. The following steps are used in the linear search algorithm:

Linear_ Search(A[ ], X)
Step 1: Initialize i to 1
Step 2: if i exceeds the end of an array then print "element not found" and Exit
Step 3: if A[i] = X then Print "Element X Found at index i in the array" and Exit
Step 4: Increment i and go to Step 2

We are given with a list of items. The following table shows a data set for linear search:

7	17	3	9	25	18

In the above table of data set, start at the first item/element in the list and compared with the key. If the key is not at the first position, then we move from the current item to next item in the list sequentially until we either find

what we are looking for or run out of items i.e the whole list of items is exhausted. If we run out of items or the list is exhausted, we can conclude that the item we were searching from the list is not present.

The key to be searched=25 from the given data set

In the given data set key 25 is compared with first element i.e7 , they are not equal then move to next element in the list and key is again compared with 17 , key 25 is not equal to 17. Like this key is compared with element in the list till either element is found in the list or not found till end of the list. In this case key element is found in the list and search is successful.

Let us write the algorithm for the linear search process first and then analyze its complexity.

```
// a is the list of n elements, key is an element to be searched in the list function
linear_search(a,n,key)
{
        found=false // found is a boolean variable which will store either true or
false
        for(i=0;i<n;i++)
        {
                if (a[i]==key)
                        found = true
                       break;
        }
        if (i==n)
                found =
false
      return found
}
```

For the complexity analysis of this algorithm, we will discuss the following cases:

a. best case time analysis
b. worst-case time analysis
c. average case time analysis

To analyze searching algorithms, we need to decide on a basic unit of computation. This is the common step that must be repeated in order to solve the problem. For searching, comparison operation is the key operation in the algorithm so it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. If the item is not in the list, the only way to know it is to compare it against every item present.

Best Case: The best case - we will find the key in the first place we look, at the beginning of the list i.e the first comparison returns a match or return found as

true. In this case we only require a single comparison and complexity will be O(1).

Worst Case: In worst case either we will find the key at the end of the list or we may not find the key until the very last comparison i.e *nth* comparison. Since the search requires n comparisons in the worst case, complexity will be O(n).

Average Case: On average, we will find the key about halfway into the list; that is, we will compare against *n*/2 data items. However, that as *n* gets larger, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the linear search, is *O*(*n*). The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility.

If the performance of the system is crucial, i.e. it's part of a life-critical system, and then we must use the worst case in our design calculations and complexity analysis as it tends to the best guaranteed performance.

The following table summarizes the above discussed results.

Case	Best Case	Worst Case	Average Case
item is present	O(1)	O(n)	O(n/2) = O(n)
item is not present	O(n)	O(n)	O(n)

However, we will generally be most interested in the worst-case time calculations as worst-case times can lead to guaranteed performance predictions.

Most of the times an algorithm run for the longest period of time as defined in worst case. Information provide by best case is not very useful. In average case, it is difficult to determine probability of occurrence of input data set. Worst case provides an upper bound on performance i.e the algorithm will never take more time than computed in worse case. So, the worst-case time analysis is easier to compute and is useful than average time case.

3.4.4 SORTING

Sorting is the process of arranging a collection of data into either ascending or descending order. Generally the output is arranged in sorted order so that it can be easily interpreted. Sometimes sorting at the initial stages increases the performances of an algorithm while solving a problem.

Sorting techniques are broadly classified into two categories:

- Internal Sort: - Internal sorts are the sorting algorithms in which the complete data set to be sorted is available in the computer's main memory.

- **External Sort:** - External sorting techniques are used when the collection of complete data cannot reside in the main memory but must reside in secondary storage for example on a disk.

In this section we will discuss only internal sorting algorithms. Some of the internal sorting algorithms are bubble sort, insertion sort and selection sort. For any sorting algorithm important factors that contribute to measure their efficiency are the size of the data set and the method/operation to move the different elements around or exchange the elements. So counting the number of comparisons and the number of exchanges made by an algorithm provides useful performance measures. When sorting large set of data, the number of exchanges made may be the principal performance criterion, since exchanging two records will involve a lot of time.

Bubble Sort

It is the simplest sorting algorithm in which each pair of adjacent elements is compared and exchanged if they are not in order. This algorithm is not recommended for use for a bigger size array because its average and worst case complexity are of $O(n^2)$ where **n** is the number of elements in an array. This algorithm is known as **bubble sort**, because the largest element in the given unsorted array, bubbles up towards the last place in every cycle/pass .

. A list of numbers is given as input that needs to be sorted. Let us explain the process of sorting via bubble sort with the help of following Tables:

First Pass

23	18	15	37	8	11
18	23	15	37	8	11
18	15	23	37	8	11
18	15	23	37	8	11
18	15	23	8	37	11
18	15	23	8	11	**37**

Second Pass

18	15	23	8	11	**37**
15	18	23	8	11	**37**
15	18	23	8	11	**37**
15	18	8	23	11	**37**
15	18	8	11	**23**	**37**

15	18	8	11	**23**	**37**
15	18	8	11	**23**	**37**
15	8	18	11	**23**	**37**
15	8	11	**18**	**23**	**37**

Third Pass

15	8	11	**18**	**23**	**37**
8	15	11	**18**	**23**	**37**
8	11	**15**	**18**	**23**	**37**

Fourth Pass

8	11	**15**	**18**	**23**	**37**
8	**11**	**15**	**18**	**23**	**37**
		Fifth Pass			
8	**11**	**15**	**18**	**23**	**37**

In this the given list is divided into two sub list sorted and unsorted. The largest element is bubbled from the unsorted list to the sorted sub list. After each iteration/pass size of unsorted keep on decreasing and size of sorted sub list gets on increasing till all element of the list comes in the sorted list. With the list of n elements, n-1 pass/iteration are required to sort. Let us discuss the result of iteration shown in above tables.

In pass 1, first and second element of the data set i.e 23 and 18 are compared and as 23 is greater than 18 so they are swapped. Then second and third element will be compared i.e 23 and 15, again 23 is greater than 15 so swapped. Now 23 and 37 is compared and 23 is less than 37 so no swapping take place. Then 37 and 8 is compared and 37 is greater than 8 so swapping take place. At the end 37 is compared with 11 and again swapped. As a result largest element of the given data set i.e 37 is bubbled at the last position in the array. At each pass the largest element among the remaining elements in the unsorted array bubbles up towards the sorted part of the array as shown in the table above. This process will continue till n-1 passes.
The first version of the algorithm for above sorting method is as below:
Bubble Sort Algorithm- Version1

```
   // A is the list of n elements to be
   sorted function bubble sort (A,n)
{
        int i,j
          for ( i= 1 to n-1
            for (j = 0 to n-2
                        {
                                if (A[j]>A[j+1])
                                {
                        // swapping of two adjacent elements of an array A
                        exchange (A[j], A[j+1])

                                }
                         }
        }
```

Let us do complexity analysis of bubble sort algorithm:

We will perform the **worst case analysis** of the algorithm. Assume that in the worst case scenario the exchange operation inside the loop will take constant amount of time (O(1)). There are two loops in the algorithm. Both the outer and inner loops will execute n-1 times. Therefore the total running time T(n) will be:

T(n) = (n-1)(n-1)* C(constant time required for simple statements like exchange)

T(n) = Cn^2 -2Cn + 1

If the worst time (big oh) complexity of the algorithm is expressed in polynomial expression, we consider(i) the highest order in the expression and (ii) do not consider the constant value in the final value. Therefore

T(n)= O(n^2)

Let us reanalyze the above algorithm to improve the running time algorithm further. From the example it is visible that the Bubble sort algorithm divides the array into unsorted and sorted sub-arrays. The inner loop rescans the sorted sub- array in each cycle, although there will not be any exchange of adjacent elements. The modified version (version 2) of the algorithm overcomes this problem:

Version -2

```
function Bubble Sort(A,n)

{

int i,j
        for ( i= 1 to n-1)
          for (j = 0 to n-i-1)
                    {
                              if(A[j]>A[j+1])
                              {
                    // swapping of two adjacent elements of an array A
                    exchange(A[j], A[j+1])

                              }
                     }
        }
```

There will be no change in the number of iterations i.e. n-2 iterations in the first pass, but in the second pass it will be n-3, in the third pass it will be n-4 iterations and so on. In this case too, the complexity remains to be O(n^2) but the number of exchange operations will be less. This requires further improvement of the algorithm.

In some cases there is no need of running n-1 passes in the outer loop. The array might be sorted in less than that. The following is the modified version (Version -3) of the algorithm

function Bubble Sort(A,n)

```
{

int i,j
        for ( i= 1 to n-1)
{
        flag = 0;
           for (j = 0 to n-i-1)
                        {
                                if(A[j]>A[j+1])
                                {
                        // swapping of two adjacent elements of an array A
                        exchange( A[j], A[j+1])
                         flag = 1;
                                }
        if (flag = = 0)
exit;
}
        }
```

In case there is no swapping, flag will remain set to 0 and the algorithm will stop running.

Time Complexity

In the modified algorithm, the inner loop will execute at least once to verify that the array is sorted but not (n-i-1) times. Therefore the time complexity will be:

$T(n) = C^* (n-1)$

$= O(n)$

3.5 SUMMARY

In this unit after making a brief review of asymptotic notations, complexity analysis of simple algorithms in illustrated simple summation, matrix multiplication, polynomial evaluation, searching and sorting. Horner's rule is discussed to evaluate the polynomial and its complexity is O(n). Basic matrix multiplication is explained for finding product of two matrices of order n*n with time complexity in the order of $O(n^3)$. For exponent evaluation both approaches i.e left to right binary exponentiation and right to left binary exponentiation is illustrated. Time complexity of these algorithms to compute x^n is O(log n).

Different versions of bubble sort algorithm are presented and its performance

analysis is done at the end.

3.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1. A polynomial is an expression that contains more than two terms. A term comprises of a coefficient and an exponent.
   Example: $P(x) = 15x^3+7x^2+9x+7$

   general form of a polynomial of degree n is

   $P(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots..+a_1x^1+a_0x^0$

2. Show the steps of Horner's rule for $p(x)= 3x_4+2x_3-5x+7$ at x=2
   poly=0 , array a[5]={7,-5,0,2,3}

   Iteration 1,
   poly = x * 0 + a[4] = 3

   Iteration 2,
   poly = x * 3 + a[3] = 2 * 3 + 2 = 6 +2 = 8

   Iteration 3,
   poly = x * 8 + a[2]
   = 2 * 8 + 0 = 16 + 0 = 16

   Iteration 4,
   poly = x * 16 + a[1]
   = 2 * 16 + (-5) = 32 -5 = 27

   Iteration 5,
   poly = x * 27 + a[0]
   = 2 * 27 + 7 = 54 + 7 = 61

3. **A basic (general) algorithm:**

   /* a is an array with polynomial coefficient, n is degree of polynomial, x is the point at which polynomial will be evaluated */

```
function(a[n], n, x)

{
        poly = 0;

        for ( i=0; i<= n; i++)

        {

                result =1;
                for (j=0; j<i; j++)

                {
```

```
                result= result * x;
            }

            poly= poly + result *a[i];

        }
        return poly.
    }
```

Time Complexity of above basic algorithm is O(n^2) where n is the degree of the polynomial. Time complexity of the Horner's rule algorithm is O(n) for a polynomial of degree n. Basic algorithm is inefficient algorithm in comparison to Horner's rule method for evaluating a polynomial.

Check Your Progress 2

1. C program to find two matrices A[3x3] and B[3x3]

```
#include<stdio.h>
int main()

{
int a[3][3],b[3][3],c[3][3],i,j,k,sum=0;

printf("\nEnter the First matrix->");
for(i=0;i<3;i++)
        for(j=0;j<3;j++)
                scanf("%d",&a[i][j]);
printf("\nEnter the Second matrix->");
for(i=0;i<3;i++)
        for(j=0;j<3;j++)
                scanf("%d",&b[i][j]);
printf("\nThe First matrix is\n");
for(i=0;i<3;i++)
{
        printf("\n");
        for(j=0;j<3;j++)
        {
                printf("%d\t",a[i][j]);
        }
}
printf("\nThe Second matrix is\n");
for(i=0;i<3;i++)


{
        printf("\n");
        for (j=0;j<3;j++)
                printf("%d\t",b[i][j]);
}
for(i=0;i<3;i++)
        for(j=0;j<3;j++)
                c[i][j]=0;

for(i=0;i<3;i++)
```

```
{
        for(j=0;j<3;j++)
        {
        sum=0;
                for(k=0;k<3;k++)
                        sum=sum+a[i][k]*b[k][j];
                c[i][j]=sum;
        }
}
printf("\nThe multiplication of two matrix is\n");
for(i=0;i<3;i++)
{
        printf("\n");
        for(j=0;j<3;j++)
                printf("%d\t",c[i][j]);
}
return 0;
}
```

Check Your Progress 3

1. Left to right binary exponentiation for a^{283} is as follows:
n=283, binary equivalent to binary string 100011011, s=9 (length of binary string)
result = a

Iteration no.	i	Bit	result
1	7	0	a2
2	6	0	a4
3	5	0	a8
4	4	1	$(a_8)_2$*a= a_{17}
5	3	1	$(a_{17})_2$*a= a_{35}
6	2	0	$(a_{35})_2$= a_{70}
7	1	1	$(a_{70})_2$*a= a_{141}
8	0	1	$(a_{141})_2$*a= a_{283}

Right to left binary exponentiation for $a^{283}$is as follows: n=283, binary equivalent to binary string 100011011, s=9 (length of binary string)
result = a (since A[0]=1)

Iteration no.	i	Bit	x	result
1	1	1	a2	a *a_2 = a_3
2	2	0	a4	a3
3	3	1	a8	a3 * a8= a11
4	4	1	(a8)2	a16 * a11= a27
5	5	0	(a16)2	(a27)
6	6	0	(a32)2	a27
7	7	0	(a64)2	a27
8	8	1	(a128)2	(a256)* a27= a283

UNIT 4 SOLVING RECURRENCES

Structure

4.0 INTRODUCTION

Complexity analysis of iteration algorithms is much easier as compared to recursive algorithms. But, once the recurrence relation/equation is defined for a recursive algorithm, which is not difficult task, then it becomes easier task to obtain the asymptotic bounds (θ, O) for the recursive solution. In this unit we focus on recursive algorithms exclusively. Three techniques for solving recurrence equation are discussed: (i) Substitution method (ii) Recursion Tree Method and Master Method. In the substitution method, we first guess an asymptotic bound and then we prove whether our guess is correct or not. In the recursion tree method a recurrences equation is converted into a recursion tree comprising several levels. Calculating time complexity requires taking a total sum of the cost of all the levels. The master method requires memorization of three different types of **cases** which help to obtain asymptotic bounds of many simple recurrence relations.

4.1 OBJECTIVES

After going through this unit you will be able to

- Define recurrence relation
- Construct recurrence relation of simple recursive algorithms
- List the techniques used to solve recurrence relation
- Solve the recurrence relation through , Substitution, Recurrence tree & Master methods.

4.2 RECURRENCE RELATION

We often use a *recurrence relation* to describe the running time of a recursive algorithm. A *recursive algorithm* can be defined as an algorithm which makes a recursive call to itself with smaller data size. Many problems are solved

recursively, especially those problems which are solved through Divide and Conquer technique. In this case, the main problem is divided into smaller sub-problems which are solved recursively. Quick Sort, Merge Sort, Binary search, Strassen's multiplication algorithm are formulated as recursive algorithms .These problems will be taken up separately in the next block.

Like all recursive functions, a recurrence relation also consists of two steps: (i) one or more initial conditions and (ii) recursive definition of a problem

Example 1: A Fibonacci sequence $f_0, f_1, f_2, \ldots\ldots$ can be defined by the recurrence relation as:

$$f_n = \begin{cases} 0 & if\ n = 0 \\ 0 & if\ n = 1 \\ f_{n-1} + f_{n-2} & if\ n \geq 2 \end{cases}$$

1. (**Basic Step**)The given recurrence says that if n=0 then $f_0 = 0$ and if n=1 then $f_1 = 1$. These two conditions (or values) where recursion does not call itself is called an **initial condition** (or **Base conditions**).
2. (**Recursive step**): This step is used to find new terms $f_2, f_3, \ldots\ldots$, from the existing (preceding) terms, by using the formula $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$.

This formula says that "by adding two previous sequence (or term) we can get the next term".

For example $\boldsymbol{f_2 = f_1 + f_0 = 1 + 0 = 1}$;
$f_3 = f_2 + f_1 = 1 + 1 = 2$; $f_4 = f_3 + f_2 = 2 + 1 = 3$ and so on

Example 2 Find out the value of n! = n (n-1) (n-2)……. (3) (2) (1) for n ≥ 1

Factorial function is defined as:

$$n! = \begin{cases} 1 & if\ n = 1 \\ n.(n-1)! & if\ n > 1 \end{cases}$$

Let us write an algorithm for factorial function:

int fact(int n)

1: $if\ n == 0\ then$
2: $return\ 1$
3: $else$
4: $return\ n * (n-1)$
5: $endif$

Let us try to understand the efficiency of the algorithm in terms of the number of multiplications operations required for each value of n

Let (n) denoted the number of multiplication required to execute the n!,

that is $T(n)$ denotes the number of times the line 4 is executed in factorial algorithm.

- We have the initial condition T(0) = 1; since when n = 0, fact simply returns (i.e. Number of multiplication is0).

- When $n > 1$, the line 4 performs 1 multiplication plus fact is recursively called with input (n -1). It means, by the definition of (n), additional $(n-1)$ number of multiplications are required.

We can write a recurrence relation for the factorial as:

$$T(n) = \begin{cases} 1, if\ n = 0 \\ 1 + (n-1) \end{cases}$$

Algorithm 3: The following algorithm calculates x^n:

Algorithm 3:*Power* (x, n)

1: **if**$(n == 0)$

2: $return\ 1$

3: $if(n == 1)$

4: $return\ x$

5: ***else***

6: $return\ x * (x, n-1);$

7: $endif$

The base case is reached when n == 0 or n== 1.

The algorithm 3 performs one comparison and one return statement. Therefore,

$(0) and\ (1) = O(1) = a$

When n > 1; the **algorithm3** performs one recursive call with input parameter (n – 1) at line 6, and some constant number of basic operations. Thus we obtain the recurrence relation as:

$$(n) = \begin{cases} (1) = a & (base\ case) \\ (n-1) + b & (Recursive\ step) \end{cases}$$

Example 4: A customer makes an investment of Rs. 5000 at 15 percent annual compound interest. If T_n denotes the amount earned at the end of n years, define a recurrence relation and initial conditions

Ans- At the end of n-1 years, the amount is T_{n-1}. After one more year , the amount will be T_{n-1} + the interest amount. Therefore

$T_n = T_{n-1} + (15\%)_{-1} = (0.15)T_{n-1} = (1.15)T_{n-1}$; n≥ 1

To find out the recurrence relation when n=1 (base value) we have to find the value of T_0.

Since T_0 refers to the initial amount, $T_0 = 5000$

With the above definitions we can calculate the value of T_n for any value of n. For example:

$T_3 = (1.15)_2 = (1.15)(1.15)T_1 = (1.15)(1.15)(1.15)T_0 = (1.15)^3(5000)$

The above computation can be extended to any arbitrary value of n.

$T_n = (1.15)_{-1}$

.. ..

$= ((1.15)(5000)$

4.3 METHODS FOR SOLVING RECURRENCE RELATIONS

Three methods are discussed here to solve recurrence relations: Substitution method, Recursion Tree method and Mater method. .We start with Substitution method

4.3.1 Substitution Method

Substitution is opposite of induction .We start at n and move backward. A substitution method is one, in which we guess a bound and then use mathematical induction to prove whether our guess is correct or not?. It comprises two steps:

Step1: Guess the asymptotic bound of the Solution.

Step2: Prove the correctness of the guess using Mathematical Induction.

Example 1. Solve the following recurrence by using substitution method.

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

Solution*: step1:* The given recurrence is quite similar to that of Merge Sort algorithm, Therefore, our guess to the solution is $(n) = O\ (nlogn)$

Or $(n) \leq c.nlogn$

Step2: Now we use mathematical Induction.

Here our guess does not hold for n=1because $(1) \leq c.1log1$
$i.e.(n) \leq 0\ which\ is\ contradiction\ with\ (1) = 1$

Now for n=2
$(2) \leq c.2log22$

$2T\left(\frac{2}{2}\right) + 2 \leq c.2$

$2(1) + 2 \leq c.2$
$0 + 2 \leq c.2$

$2 \leq c.2$ which is true. So $(2) \leq c.nlogn$ is True for n = 2
So
$(2) \leq c.nlogn$ is True form = 2

(i) **Induction step:** Now assume it is true for n = n/2

Now we have to show that it is true for the value of n

$i.e.(2) \leq c.nlogn$

We known that $(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + n$

$\leq 2(c \lfloor \frac{n}{2} \rfloor log \lfloor \frac{n}{2} \rfloor + n$

$\leq cnlog \lfloor \frac{n}{2} \rfloor + n \leq cnlogn - cnlog2 + n$

$\leq cnlogn - cn + n$

$\leq cnlogn \; \forall c \geq 1$

Thus $(n) = (nlogn)$

Remark: Making a good guess, which can be a solution of a given recurrence, requires experiences. So, in general, we are often not using this method to get a solution of the given recurrence.

4.3.2 RECURSION TREEMETHOD

A recursion tree is a convenient way to visualize what happens when a recurrence is iterated. It is a pictorial representation of a given recurrence relation, which shows how Recurrence is divided till Boundary conditions.

Recursion tree method is especially used to solve a recurrence of the form:

$T(n) = aT(\frac{n}{b}) + (n) \ldots\ldots\ldots (1) \qquad where\ a > 1, b \geq 1$

This recurrence (1) describe the running time of any divide-and-conquer algorithm.

Method (steps) for solving a recurrence $(n) = aT(\frac{n}{b}) + (n)$ using recursion tree:

We make a recursion tree for a given recurrence as follows:

a) To make a recursion tree of a given recurrence (1), First put the value of $f(n)$ at root node of a tree and make a ***a number*** of child nodes of this root value $f(n)$ Now tree will be looks like as:

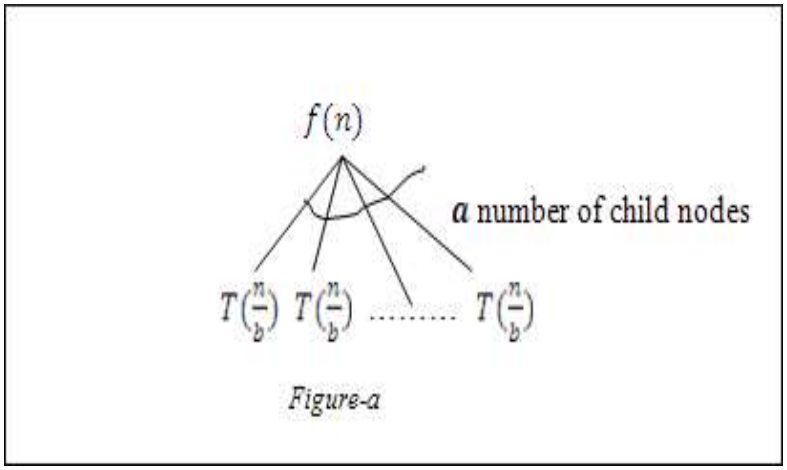


Figure-a

(b) Now we have to find the value of $\boldsymbol{T}\left(\frac{\boldsymbol{n}}{\boldsymbol{b}}\right)$ by putting (n/b) in place of n in equation (1).That is

$$T\left(\frac{n}{b}\right) = aT\left(\frac{\frac{n}{b}}{b}\right) + f(n/b) = aT + f\left(\frac{n}{b^2}\right) + f(n/b) \ldots (2)$$

From equation (2), now (n/b) will be the value of node having a branch (child nodes) each of size T(n/b). Now each $T\left(\frac{n}{b}\right)$in **figure-a** will be replaced as follows:

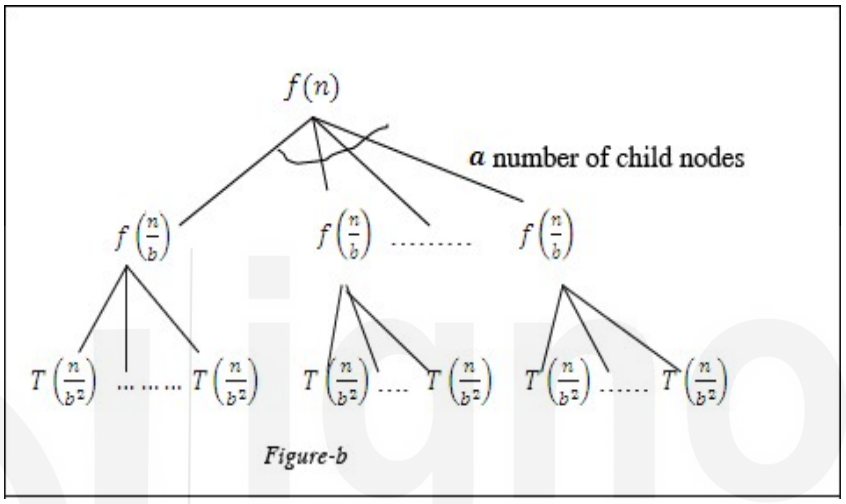


Figure-b

c) In this way you can expend a tree one more level (i.e. up to (at least) 2 levels).

Step2: (a) Now you have to find per level cost of a tree. Per level cost is the sum of the cost of each node at that level. For example per level cost at level1 is

$\left(\frac{n}{b}\right) + f\left(\frac{n}{b}\right) + \cdots f\left(\frac{n}{b}\right)$ $(a\ times)$. This is also called **Row-Sum.**

(b) Now the total (final) cost of the tree can be obtained by taking the sum of costs of all these levels.

$i.e. Total\ cost = sum of\ costs\ of\ (l_0 + l_1 + \cdots \ldots \ldots . + l_k)$.

This is also called **Column-Sum.**

Let us take one example to understand the concept to solve a recurrence using recursion tree method:

Example1: Solve the recurrence $(\boldsymbol{n}) = \boldsymbol{2T}\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}\right) + \boldsymbol{n}$ using recursion tree method.

Solution: **Step1:** First you make a recursion tree of a given recurrence.

1. To make a recursion tree, you have to write the value of (n) at root node. And
2. The number of child of a Root Node is equal to the value of a. (Here the value of a = 2).So recursion tree be looks like as:

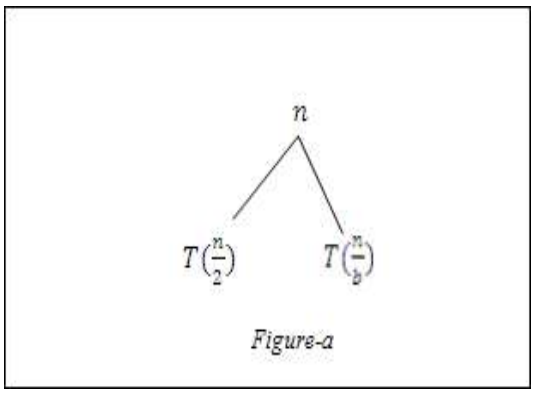


Figure-a

b) Now we have to find the value of $T\left(\frac{n}{2}\right)$ in figure (a) by putting (n/2) in place of n in equation (1).That is

$$T\left(\frac{n}{b}\right) = 2T\left(\frac{\frac{n}{2}}{2}\right) + n = 2T\left(\frac{n}{2^2}\right) + n/2 \ldots (2)$$

From equation (2), now $\left(\frac{n}{2}\right)$ will be the value of node having 2 branch (child nodes) each of size T(n/2). Now each $T\left(\frac{n}{2}\right)$ in **figure-a** will be replaced as follows:

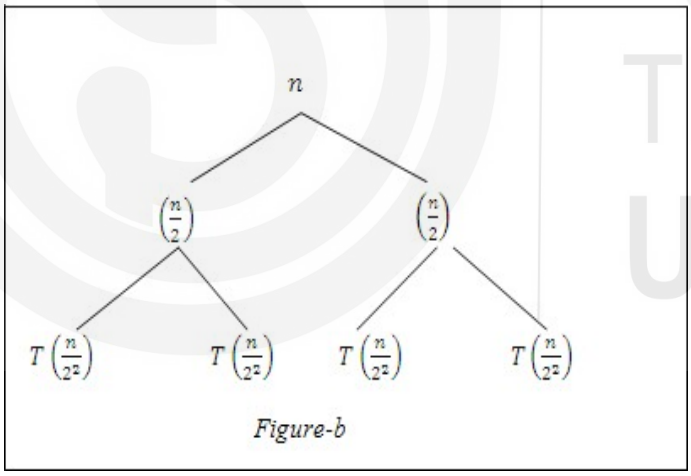


Figure-b

c) **In this way, you can extend a** tree up to Boundary condition (when problem size becomes 1). So the final tree will be looks like:

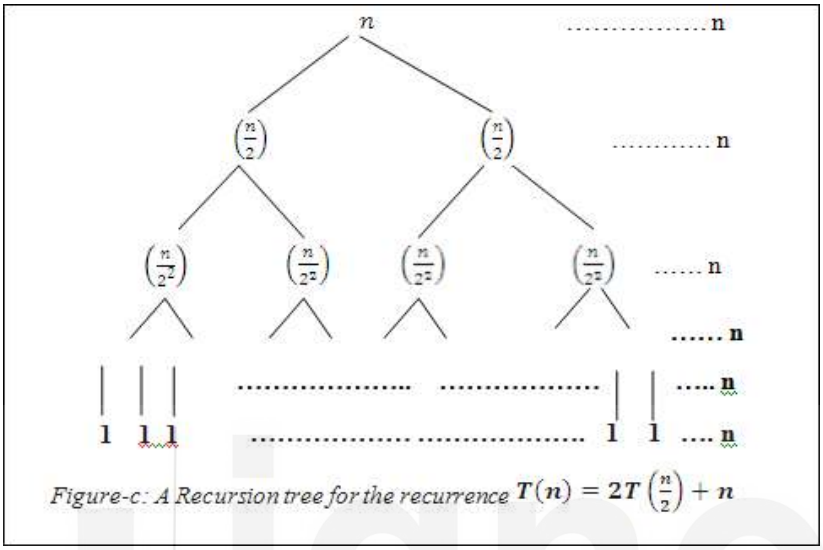


Figure-c: A Recursion tree for the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + n$

Now we find the per level cost of a tree, Per-level cost is the sum of the costs within each level (called row sum). Here per level cost is For example: per level cost at depth 2 in **figure-c** can be obtained as:

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

Then total cost is the sum of the costs of all levels (called column sum), which gives the solution of a given Recurrence. The height of the tree is

Total cost= $n + n + n + \cdots \ldots \ldots + n$.....(3)

To find the sum of this series you have to find the total number of terms in this series. To find a total number of terms, you have to find a height of a tree.

Height of tree can be obtained as follow (see recursion tree of figure c): you start a problem of size n, then problem size reduces to$(\frac{n}{2})$, then$(\frac{n}{2^2})$, and so on till boundary condition (problem size 1) is not reached. That is

$$n \to \left(\frac{n}{2}\right) \to \left(\frac{n}{2^2}\right) \to \cdots \ldots \ldots \ldots \to \left(\frac{n}{2^k}\right)$$

At last level problem size will be equal to 1 if

$$\left(\frac{n}{2^k}\right) = 1 \Rightarrow n = 2^k \Rightarrow k = log_2 n.$$

This k represent the height of the tree, hence height $= k = log_2 n$.

Hence total cost in equation (3) is

$$n + n + n + \cdots \ldots \ldots + (log_2 n \; terms) = nlog_2 n \Longrightarrow (nlog_2 n).$$

Example2: Solve the recurrence $T(n) = T(\lfloor \frac{n}{3} \rfloor) + T(\lfloor \frac{2n}{3} \rfloor) + n$ using recursion tree method.

Solution: We always omit floor & ceiling function while solving recurrence. Thus given recurrence can be written as:

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

Figure-a to figure-c shows a step-by-step derivation of a recursion tree for the given recurrence (1).

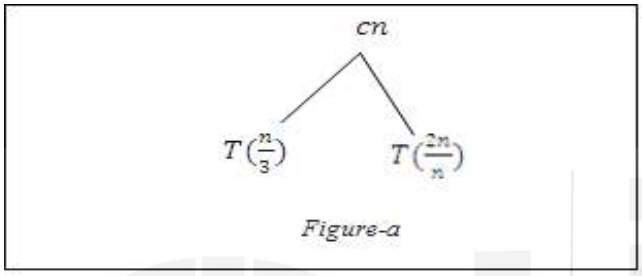


Figure-a

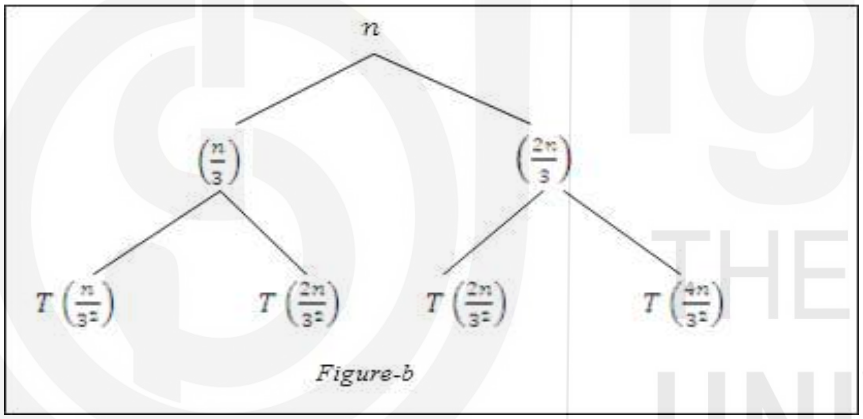


Figure-b

c) **In this way, you can extend a** tree up to Boundary condition (when problem size becomes 1). So the final tree will be looks like:

Here the smallest path from root to the leaf is:

$$n \to \left(\frac{2}{3}\right)n \to \left(\frac{2}{3}\right)^2 n \to \cdots 1$$

$$\left(\frac{2}{3}\right)^k = 1 \Longrightarrow k = log_{3/2}n \Longrightarrow \textit{Height of the tree.}$$

$$n + n + \cdots\ldots\ldots\ldots + n\left(log_{3/2}n \; times\right)$$

$$\Longrightarrow nlog_{3/2}n = \frac{nlog_2 n}{log_2 \frac{3}{2}} = O(nlog_2 n) - - - - - (*)$$

Here the smallest path from root to the leaf is:

$n \to (\frac{n}{3}) \to (\frac{n}{3^2}) \to \cdots \ldots \ldots \ldots \to (\frac{n}{3^k})$

$(n/3)^k = 1 \Longrightarrow k = log_3 n \Longrightarrow Height\ of\ the\ tree.$

$n + n + \cdots \ldots \ldots \ldots + (log_3 n\ times)$

$$\Longrightarrow nlog_3 n = \frac{nlog_2 n}{log_2 3} = \Omega(nlog_2 n) - - - - - (**)$$

Form equation (*) and
(**), $Since\ T(n) = O(nlog_2 n) and\ T(n) = \Omega(nlog_2 n)$, thus we write:

$T(n) = \Theta(nlog_2 n)$

Remark: If
$\boldsymbol{f(n) = O(g(n)) and\ f(n) = \Omega(g(n)) then\ fIn) = \Theta(g(n))}$

Example3: A recurrence relation for Tower of Hanoi (TOH) problem is $T(n) = 2T\ (n-1) + 1\ with\ (1) = 1\ and\ T(n) = 3$. Solve this recurrence to findthe solution of TOH problem.

Solution:

Figure-a to figure-c shows a step-by-step derivation of a recursion tree for the given recurrence $T(n) = 2T\ (n-1) + 1$

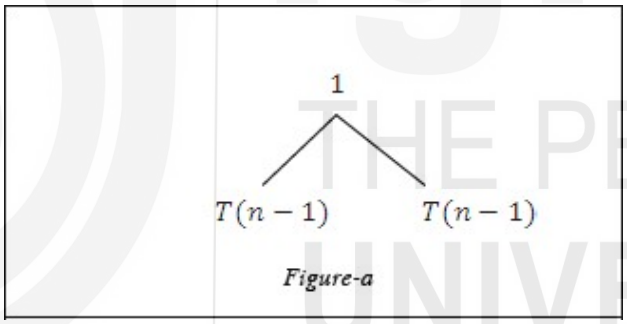


Figure-a

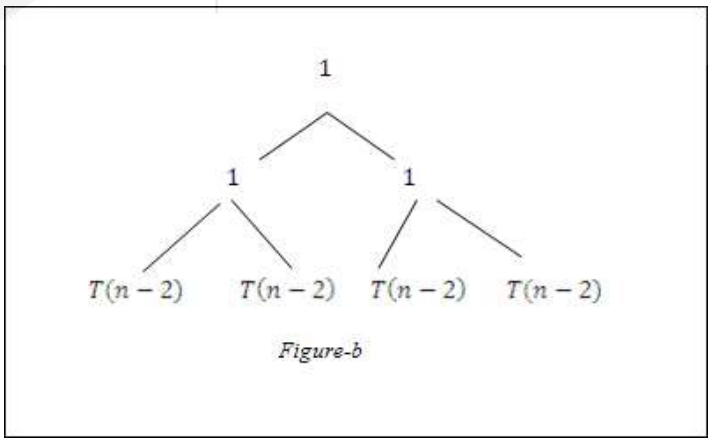


Figure-b

c) In this way, you can extend a tree up to Boundary condition (when problem size becomes 1). That is

$n \to (n-1) \to (n-2) \to \cdots \ldots \ldots . \to (n - (n-1))$

$\Longrightarrow n \to (n-1) \to (n-2) \to \cdots \ldots \ldots \ldots \to 2 \to 1$

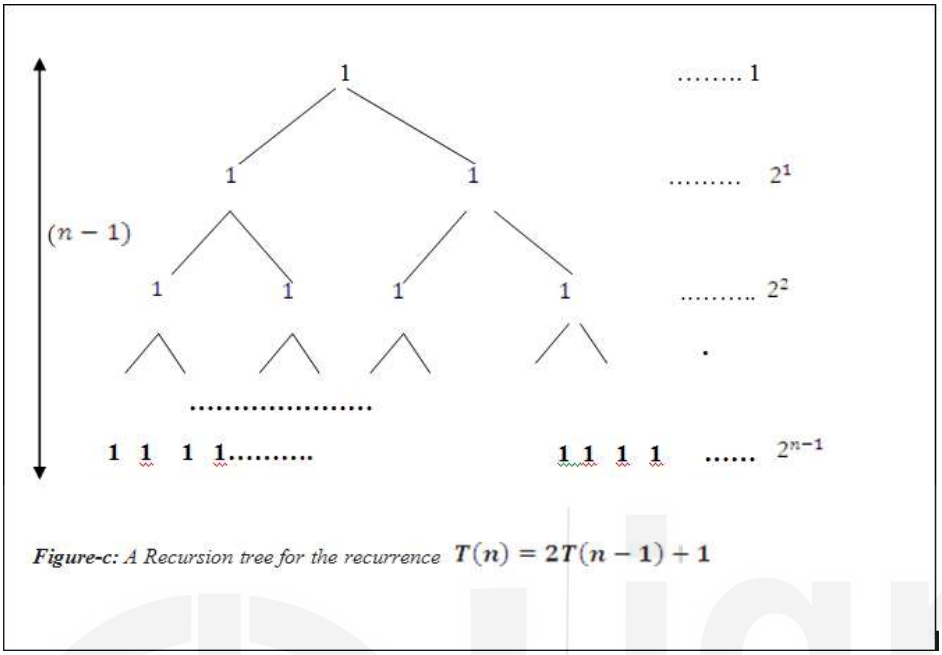


Figure-c: A Recursion tree for the recurrence $T(n) = 2T(n-1) + 1$

So the final tree will be looks like:

At last level problem size will be equal to 1 if
$(n-(n-1)) = 1 \Rightarrow Hight\ of\ the\ tree \Rightarrow (n-1)$.

Hence Total Cost of the tree in figure (c) can be obtained by taking column sum upto the height of the tree.

$$T(n) = 1 + 2^1 + 2^2 + \cdots \ldots + 2^{n-1} = \frac{1(2^n - 1)}{2 - 1} = 2^n - 1.$$

Hence the solution of TOH problem is $T(n) = (2^n - 1)$

☞ **Check Your Progress 1**

Q1: write a recurrence relation for the following recursive functions:

a) *Fast_Power*(x, n)

{*if* $(n == 0)$

return 1;

elseif $(n == 1)$

return x;

elseif $((n\%2) == 0)$ *//if n is even*

return Fast_power $\left(x, \frac{n}{2}\right) *$ *Fast_power* $\left(x, \frac{n}{2}\right)$;

else

$return\ x * Fast\_power\ (x, \frac{n}{2}) * Fast\_power\ (x, \frac{n}{2})$

}

b)

Fibnacci (n)

{ if (*n* = = *0*)

return 0;

if (*n* = = *1*)

return 1;

return fibnacci (*n*-1) + *fibnacci* (*n*-2); }

Q.2: Solve the following recurrence Using Recursion tree method

a. $T(n) = 4T\ (\lfloor \frac{n}{2} \rfloor) + n$

b. $T(n) = T\ (\frac{n}{2}) + T\ (\frac{n}{4}) + T\ (\frac{n}{8}) + n$

4.3.3 MASTERMETHOD

Definition 1: A function *f*(*n*) is asymptotically positive if any only if there exists a real number *n* such that f(*x*) > 0 for all *x* > *n*.

The master method provides us a straight forward method for solving recurrences of the form $\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$, where *a* ≥ 1 and *b* > 1 are constants and *f(n)* is an asymptotically positive function. This recurrence gives us the running time of an algorithm that divides a problem of size *n* into *a* **subproblems** of size $\left(\frac{n}{b}\right)$.

The *a* **subproblems** are solved recursively, each in time $T\left(\frac{n}{b}\right)$. The cost of dividing the problem and combining the results of the subproblems is described by the function *f(n)* This recurrence is technically correct only when $\left(\frac{n}{b}\right)$ is an integer, so the assumption will be made that $\left(\frac{n}{b}\right)$ is either $\left\lfloor\frac{n}{b}\right\rfloor$ or $\left\lceil\frac{n}{b}\right\rceil$ since such a replacement does not affect the asymptotic behavior of the recurrence.

The value of *a* and *b* is a positive integer since one can have only a whole number of subproblems.

Theorem1: Master Theorem

The Master Method requires memorization of the following 3 cases; then the solution of many recurrences can be determined quite easily, often without using pencil & paper.

Let $T(n)$ be defined on the non negative integers by:
$T(n) = aT\left(\frac{n}{b}\right) + f(n), where\ a \geq 1, b > 1\ and\ \frac{n}{b}$ is treated as above ------(1)

Then $T(n)$ can be bounded asymptotically as follows:

Case1: If $f(n) = O(n^{log_b a - e})$ for some $\in > 0, then\ T(n) = \Theta(n^{log_b a})$

Case2: If$(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}.logn)$.

Case3: If $(n) = (n^{log_b a + \in})$for some $\in > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant c $< 1\ and\ all\ sufficiently\ large\ n. T(n) = \theta(f(n))$

Remark: To apply Master method you always compare $n^{log_b a}$ and $f(n)$The larger of the two functions determines solution to the recurrence problems. If the growth rate of these two functions then it belongs to **case 2**.In this case we multiply by a logarithmic factor to get the run time solution (T(n)) of recurrence relation.

If $f(n)$ is polynomially smaller than n^{log_b} (by a factor of n^{ϵ}then **case 1** will be applicable to find $T(n)$.

If $f(n)$ is polynomially larger than $n^{log_b a}$ (by a factor of$1/n^{\epsilon}$ then $T(n) = \theta(f(n)$ *which is in* ***case*** **3**.

Examples of Master Theorem

Example1: Consider the recurrence of $T(n) = 9T\left(\frac{n}{3}\right) + n$, in which a = 9, b = 3, $f(n) = n$, $n^{log_b a} = n^2$ and $f(n) = O\left(n^{log_b a - \epsilon}\right)$, where $\epsilon = 1$. The growth rate of f(n) is slower, we will apply the case 1 of Master Theorem and we get $T(n) = \Theta(n^{log_b{}^{a-\epsilon}}) = \Theta(n^2)$

Example2: Consider the recurrence of $T(n) = T\left(\frac{2n}{3}\right) + 1$, in which $a = 1$, $b = \frac{3}{2}, f(n) = n^{log_3 2}$=1. Since $f(n) = \theta(n^{log_3 2})$. By Master Theorem (case2), we get $T(n) = \Theta(n^{log_b a}.logn) = \Theta(logn)$.

☞ **Check Your Progress 2**

Q.1: Write the first two cases (Case 1 and Case 2) of Master method to solve a recurrence relation of the form:
$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

.....

.....

.....

.....

Q.2: Use Master Theorem to give the tight asymptotic bounds of the following recurrences:

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

.....

.....

.....

.....

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

4.4 SUMMARY

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation which describes a function in terms of its value on smaller inputs.

There are three basic methods of solving the recurrence relation:

1. The Substitution Method
2. The Recursion-tree Method
3. The Master Theorem

Master method provides a "cookbook" method for solving recurrences of the $T(n) = aT\left(\frac{n}{b}\right) + f(n) where\ a \geq 1\ and\ b > 1$ are constants from:

$f(n)$

In master method you have to always compare the value of $f(n)$ with $n^{\log_b a}$ to decide which case is applicable.

4.5 SOLUTIONS/ANSWERS

Check Your Progress 1:

Q 1: a)

At every step the problem size reduces to half the size. When the power is an odd number, the additional multiplication is involved. To find a time complexity of this algorithm, let us consider the ***worst case,*** that is we assume that at every step additional multiplication is needed. Thus total number of operations T(n) will reduce to number of operations for n/2, that is T(n/2) with three additional arithmetic operations(In odd power case: 2 multiplication and one division). Now we can write:

$(n) = 1\ if\ n = 0\ or\ 1$

$(n) = T(\frac{n}{2}) + 3\ if \text{n} \geq 2$

Instead of writing exact number of operations needed by the algorithm, we can use some constants. The reason for writing this constant is that we are always interested to find "asymptotic complexity" instead of finding exact number of operations needed by algorithm, and also it would not affect our complexity also.

$$T(n) = \begin{cases} T(1) = a & if\ n = 0\ or\ n = 1 \quad (base\ case) \\ T\left(\frac{n}{2}\right) + b & if\ n \geq 2 \quad (Recursive\ step) \end{cases}$$

Solving Recurrence

b) $T(n) = \begin{cases} a & if\ n = 0\ or\ n = 1 \quad (base\ case) \\ T(n-1) + T(n-2) + b & if\ n \geq 2 \quad (Recursive\ step) \end{cases}$

Q2 *(a)* The recursion tree for the given recurrence relation is:

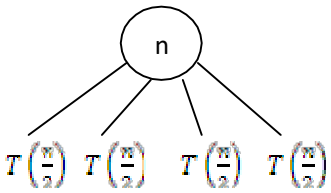


Figure a

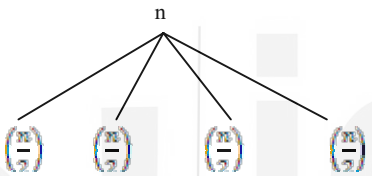


Figure b

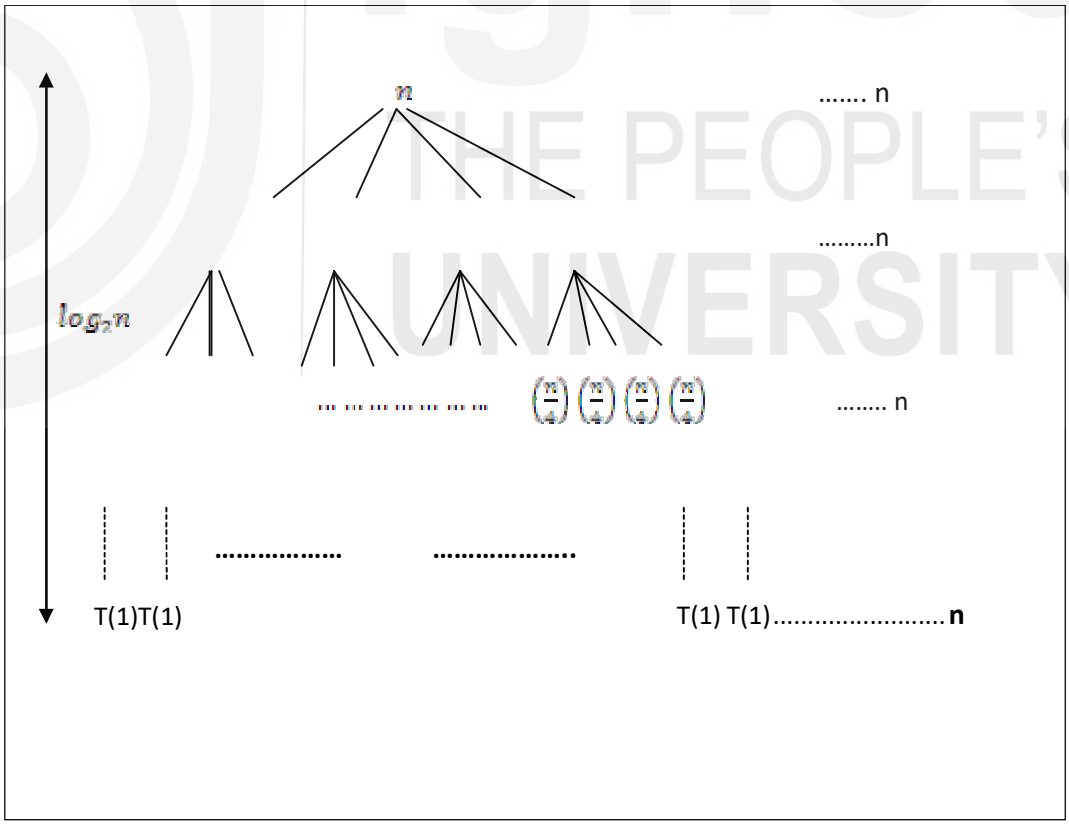


Figure c: A Recurrence Tree for $T(n) = 4T\left(\frac{n}{2}\right) + n$

We have $Total = n + 2n + 4n + \cdots log_2 n\ times$
= n (1 + 2 + 4 +… $log_2 n\ times$)

$= n\frac{(2log^2 - 1)}{(n)2 \equiv 1} = n^2 - n = (n^2)$

Q2(d)

Check Your Progress 2:

Q1: The following 3 cases are used to solve a recurrence

Case1: If for some $(n) = (n^{log_b a - \epsilon})$ for some $\epsilon > 0$,then $T(n) = \Theta(n^{log_b a})$

Case2: If $(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}. logn)$..

Solution2:

a) In a recurrence $(n) = 4T\left(\frac{n}{2}\right) + n, a = 4, b = 2,$
$(n) = n^{log_b a} = n^2. Now\ compare\ f(n) with\ n^{log_b a}.$
$Since\ f(n) = O\ (n^{log_b a - \epsilon}).$
$where \epsilon == 1. By\ Master\ Theorem\ case\ 1\ we\ get\ (n) = \Theta\ (n^2).$

b) $(n) = 4T\left(\frac{n}{2}\right) + n^2$; in which $a = 4, b = 2,\ f(n) = n^2.$

$Now\ compare\ f(n) wth\ n^{log_b a};$
$since\ (n) = {}^2 = \Theta(n^{log_b a}).$
$Thus\ By\ Master\ Theorem\ (case 2),$
$we\ get\ (n) = \Theta(n^{log_b a}. logn) = \Theta(n^2 logn).$