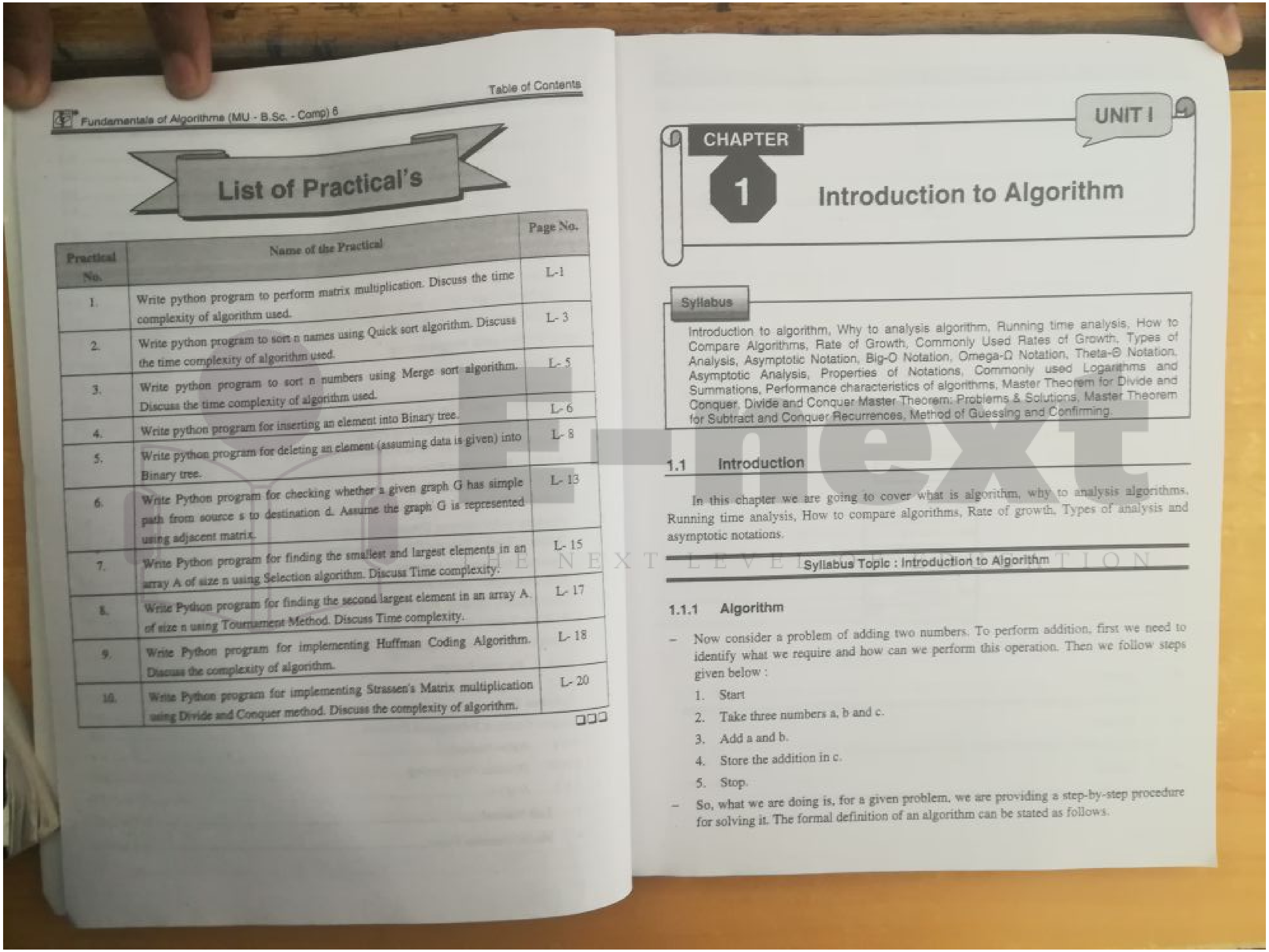


## List of Practical's

| Practical No. | Name of the Practical | Page No. |
|---|---|---|
| 1. | Write python program to perform matrix multiplication. Discuss the time complexity of algorithm used. | L-1 |
| 2. | Write python program to sort n names using Quick sort algorithm. Discuss the time complexity of algorithm used. | L- 3 |
| 3. | Write python program to sort n numbers using Merge sort algorithm. Discuss the time complexity of algorithm used. | L- 5 |
| 4. | Write python program for inserting an element into Binary tree. | L- 6 |
| 5. | Write python program for deleting an element (assuming data is given) into Binary tree. | L- 8 |
| 6. | Write Python program for checking whether a given graph G has simple path from source s to destination d. Assume the graph G is represented using adjacent matrix. | L- 13 |
| 7. | Write Python program for finding the smallest and largest elements in an array A of size n using Selection algorithm. Discuss Time complexity. | L- 15 |
| 8. | Write Python program for finding the second largest element in an array A of size n using Tournament Method. Discuss Time complexity. | L- 17 |
| 9. | Write Python program for implementing Huffman Coding Algorithm. Discuss the complexity of algorithm. | L- 18 |
| 10. | Write Python program for implementing Strassen's Matrix multiplication using Divide and Conquer method. Discuss the complexity of algorithm. | L- 20 |

❑❑❑

UNIT I

# CHAPTER 1 Introduction to Algorithm

**Syllabus**

Introduction to algorithm, Why to analysis algorithm, Running time analysis, How to Compare Algorithms, Rate of Growth, Commonly Used Rates of Growth, Types of Analysis, Asymptotic Notation, Big-O Notation, Omega-Ω Notation, Theta-Θ Notation, Asymptotic Analysis, Properties of Notations, Commonly used Logarithms and Summations, Performance characteristics of algorithms, Master Theorem for Divide and Conquer, Divide and Conquer Master Theorem: Problems & Solutions, Master Theorem for Subtract and Conquer Recurrences, Method of Guessing and Confirming.

## 1.1 Introduction

In this chapter we are going to cover what is algorithm, why to analysis algorithms, Running time analysis, How to compare algorithms, Rate of growth, Types of analysis and asymptotic notations.

**Syllabus Topic : Introduction to Algorithm**

### 1.1.1 Algorithm

- Now consider a problem of adding two numbers. To perform addition, first we need to identify what we require and how can we perform this operation. Then we follow steps given below :
  1. Start
  2. Take three numbers a, b and c.
  3. Add a and b.
  4. Store the addition in c.
  5. Stop.
- So, what we are doing is, for a given problem, we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as follows.

"An algorithm is a finite collection of well defined steps designed to solve a particular problem".

Or

"An algorithm is the step-by-step execution of a particular problem".

- An algorithm takes a set of values, as input and produces a value, or a set of values, as output.
- An algorithm may be specified as :
  - In English
  - As a computer program
  - As a pseudo-code
- An algorithm is a well-developed, organized approach to solving a complex problem.
- An algorithm is a set of instructions and instructions are steps.

  Program = Data Structure + Algorithms.

### 1.1.2 Characteristics of Algorithm

An algorithm must have following characteristics.

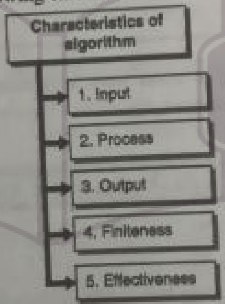


Fig. C1.1: Characteristics of algorithm

➜ **1. Input**

An algorithm must take zero or more inputs that are required for a solution of a problem.

➜ **2. Process**

An algorithm must perform certain operations on the input data.

➜ **3. Output**

An algorithm should produce certain output after processing the inputs.

➜ **4. Finiteness**

An algorithm must terminate after executing certain finite number of steps.

➜ **5. Effectiveness**

An algorithm should be simple and clear. Each step in the algorithm must be unambiguous, feasible and definite.

**Syllabus Topic : Why to Analysis Algorithm?**

## 1.2 Why to Analysis Algorithm?

Algorithms are about :

1. Solving problems.
2. Finding the better way to solve a problem without analyzing an algorithms, how do you know whether an algorithm is good or not.

- For example, To sort the numbers, we have many algorithms, like insertion sort, selection sort, quick sort, merge sort etc. But we choose the one which is effective in terms of time and speed consumed.
- In computer science, the analysis of algorithms is the determination of the amount of time, storage and/or other resources necessary to execute them.
- So, the choice of an efficient algorithm is of great importance, which can be made by considering the following factors :
  - Programming requirements of an algorithm.
  - Time requirement of an algorithm.
  - Space requirement of an algorithm.

**Syllabus Topic : Running Time Analysis**

## 1.3 Running Time Analysis

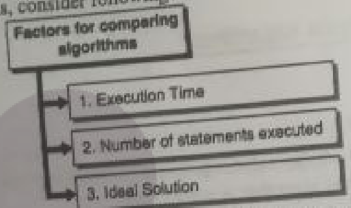
- One of the most important aspects of an algorithm is how fast it is. It is the process of determining how processing time increases as the size of the problem increases (input size). Input size is the number of elements in the input.
- It is not very easy to calculate the exact time requirements for any algorithm as it depends upon various factors like machine on which algorithms is to be executed, algorithm itself and input size of the algorithm.

- Because the processor speed in different machines may be different, So, we mainly concentrate to estimate the execution time of an algorithm irrespective to the processor/machine.

**Syllabus Topic : How to Compare Algorithms?**

## 1.4 How to Compare Algorithms?

To compare algorithms, consider following factors :



Factors for comparing algorithms
1. Execution Time
2. Number of statements executed
3. Ideal Solution

**Fig. C1.2 : Factors for comparing algorithms**

➔ **1. Execution time**

It is not a good measure as execution times are specific to a particular computer because processor speed in different computer may different.

➔ **2. Number of statements executed**

It is also not a good measure because the number of statements varies with the programming language as well as the style of the individual programmer.

➔ **3. Ideal solution**

- Let us assume that we express the running time of a given algorithm as a function of the input size n(i.e. f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time programming style etc.
- Suppose there is a statement x=x + y somewhere in the middle of the program. We wish to determine the total time that statement will spend executing, given some initial state of input data. This requires essentially two items of information, the statement frequency count(i.e. the number of times the statement will be executed) and the time for one execution. The product of these two numbers is the total time.

☞ **Example**

- Consider the three program segments (a), (b), (c).

| (a) | (b) | (c) |
|---|---|---|
| x = x + y | For i in range(1,n):<br>x = x + y | For i in range(1,n):<br>For j in range(1,n):<br>x = x + y |

- Thus for segment (a), the frequency count of this statement is 1.

  For segment (b), the count is n.

  For segment (c), the count is $n^2$.
- These frequencies 1, n, $n^2$ are said to be different, increasing order of magnitude. An order of magnitude is a common notation with which we are all familiar. For example, walking, bicycling, riding in a car and flying in an airplane represents increasing orders of magnitude with respect to distance we can travel per hour.
- In connection with algorithm analysis, the order of magnitude of a statement refers to its frequency of execution, while the order of magnitude of an algorithm refers to the sum of the frequencies of all of its statements.
- Given three algorithms for solving the same problem whose order of magnitudes are n, $n^2$, $n^3$ , naturally we will prefer the first one, since the second and third are progressively slower. For example, if n = 10 then these algorithms will require 10, 100 and 1000 units of time to execute respectively.

**Syllabus Topic : Rate of Growth**

## 1.5 Rate of Growth

- Running time analysis is the behavior of the algorithm in terms of input. How the algorithm is behaving when we keep input increased.
- Rate of growth is nothing but the representation of running time and space of an algorithm.
- We want to examine the rate of growth f(n) with respect to standard functions like log n, $n^1$, $n^2$, $n^3$, $2^n$ etc.

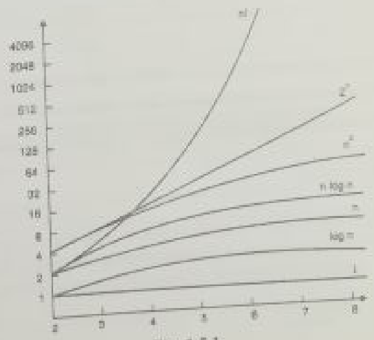


Fig. 1.5.1

**Syllabus Topic : Commonly used Rates of Growth**

## 1.6 Commonly Used Rate of Growth

The most common computing times for algorithms are as follows.

| Time Complexity | Name | Example |
|---|---|---|
| 1 | Constant | Adding an element to front of linked list. |
| log n | Logarithmic | Finding an element in a sorted array. |
| n | Linear | Finding an element in an unsorted array. |
| n log n | Linear Logarithmic | Sorting n times by divide and conquer. |
| $n^2$ | Quadratic | Shortest path between two nodes in a graph. |
| $n^3$ | Cubic | Matrix multiplication. |
| $2^n$ | Exponential | The towers of Hanoi problem. |

Here,

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$.

| n | O(log n) | O(n) | O(n log n) | $O(n^2)$ | $O(n^3)$ |
|---|---|---|---|---|---|
| 8 | 3 | 8 | 24 | 64 | 512 |
| 16 | 4 | 16 | 64 | 256 | 4096 |
| 32 | 5 | 32 | 160 | 1024 | 32768 |
| 64 | 6 | 64 | 384 | 4096 | 262144 |
| 128 | 7 | 128 | 896 | 16384 | 2097152 |
| 256 | 8 | 256 | 2048 | 262144 | 67108864 |

Rate of growth of some standard functions with the size of the input.

**Syllabus Topic : Types of Analysis**

## 1.7 Types of Analysis

- To analyze the given algorithm, we need to know with which inputs the algorithm takes less time and with which inputs the algorithm takes a long time. That means we represent the algorithm with multiple expressions : One case which takes less time and another case which takes more time.
- There are three types of analysis.

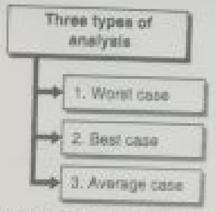


Fig. C1.3 : Types of Analysis

### ➜ 1. Worst case

If the running time of the algorithm is longest for all the inputs then it is called worst case. In this type, the key operations are executed maximum number of times.

### ➜ 2. Best case

If the running time of the algorithm is shortest for all the inputs then it is called best case. In this type, the key operations are executed minimum number of times.

### ➜ 3. Average case

If the running time of the algorithm falls between the worst case and the best case then it is called average case. To calculate the average case complexity of an algorithm, we have to take some assumptions.

☞ **Example**

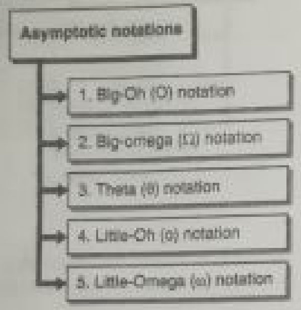
- For example, searching for an element in an unsorted array of length N.
- If we found the desired element at first position then the number of comparisons will be 1. So it is **Best case.**
- If we found the desired element at last position then the number of comparisons will be N So it is **Worst case.**
- If we assume it is found roughly in the middle portion of the array then we need to do N/2 comparisons to get the desired element, so it is **Average case.**

**Syllabus Topic : Asymptotic Notation**

## 1.8 Asymptotic Notations

- Asymptotic Notations are languages that allows us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.
- It is used to find out which algorithm is better for the same problem statement.

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm:



Asymptotic notations
1. Big-Oh (O) notation
2. Big-omega (Ω) notation
3. Theta (θ) notation
4. Little-Oh (o) notation
5. Little-Omega (ω) notation

**Fig. C1.4 : Asymptotic Notations**

- All the above notations are used to express the complexity of an algorithm.
- "Complexity can be defined as the rate at which the storage or time requirement grows as a function of the problem size."
- Complexity is the time and space requirement of the algorithm. If the time and space requirement of the algorithm is more, then complexity of the algorithm is more and if the time and space requirement of the algorithm is less, then complexity of the algorithm is less.
- Out of the two factors, time and space, the space requirement is of the algorithm is not a very important because it is available at very low cost.
- Only the time requirement of the algorithm is considered as an important factor to find the complexity. Because of the importance of time in finding the complexity, it is sometimes termed as **time complexity**.

### 1.8.1 Big-Oh(O) Notation

- Big-O notation gives the tight upper bound of the function. It measure the worst case time complexity or the longest amount of time an algorithm possibly take to complete.
- This means, a function f(x) is Big-O of function g(x) and there exists the positive constants c and $n_0$such that,

$$f(n) < = c \times g(n)$$

where $c > 0$ and $n_0 > 1$ the for all $n > = n_0$

$$f(n) = O(g(n))$$

- Here, f(x) and g(x) are the functions of the non-negative integers.

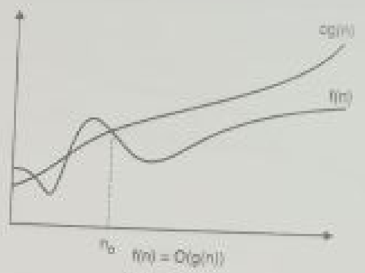


**Fig. 1.8.1**

**Example 1.8.1**

Consider the functions, f(n)=2n + 4 and g(n)=n then prove that f(n) = O(g(n)).

**Solution :**

$$f(n) < = c^*g(n) \quad \text{for } n > 0,$$

For some $c > 0$ and $n_0 > 1$, substitute $f(n) = 2n + 4$

$$2n + 4 < = c(n)$$

When can be 2n + 4 less than or equal to c(n).

For above case the values of c can be anything above 3 is better and $n_0 = 3$.

$$2n + 4 < = c(n)$$

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

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

$$4 < = n$$

It means, for every n>=4 at c=3, f(n) <= c*g(n).

### 1.8.2 Big-Omega(Ω) Notation

- Big-Ω notation gives tighter lower bound of the given function. It measures the best case time complexity or the best amount of time an algorithm possibly take to complete.
- This means, a function f(x) is Big-Ω of function g(x) and there exists the positive constants c and $n_0$such that,

$f(n) >= c \times g(n)$ where $c > 0$ and $n_0 > 1$ the for all $n >= n_0$

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

Here, f(x) and g(x) are the functions of the non-negative integers.

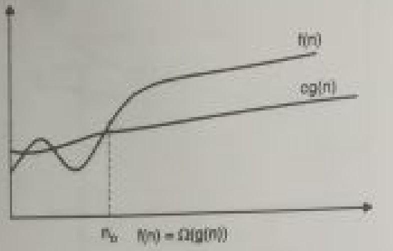


**Fig. 1.8.2**

**Example 1.8.2**

Consider the functions, f(n)=3n+2 and g(n)=n then prove that f(n) = Ω(g(n)).

**Solution :**

f(n) > = c*g(n) for n >0.

For some c > 0 and $n_0$> 1, substitute f(n) = 3n + 2

3n + 2< = c(n)

When can be 3n + 2greater than or equal to c(n),

For above case the values of c can be anything above 4 is better.

3n + 2 > = c(n)

3n + 2> = 3n, for n >= 2

Put n = 2 then 8 >= 6.

It means, 3n + 2 is lower bounded.

### 1.8.3 Theta(Θ) Notation

- Theta(Θ) notation lies between upper bound and lower bound of an algorithm. Using this we can compute the average amount of time taken by any algorithm.
- This means a function f(x) is Theta of function g(x) and there exists three positive constants c1, c2 and $n_0$such that,

  0< = c1* g(n) <= f(n) >= c2 * g(n) for all n > $n_0$

- Here, f(x) and g(x) are the functions of nonnegative integers.

  It may be noted that f(n) = Θ(g(n)) if and only if ,

f(n) = O(g(n)) and f(n) = Ω (g(n)).

- It means after some input $n_0$, the value of c1 * g(n) should be less than or equal to f(n) and c2 * g(n) should be greater than or equal to f(n).

**Example 1.8.3**

For example, let f(n)=3n+2 and g(n)=n then prove that f(n) =Θ (g(n)).

**Solution :**

As 3n+2 > = 3n for all n >=2 and

3n+2 < = 4n for all n<=2

Here, $c_1$ = 3, $c_2$ = 4 and $n_0$= 2 So f(n) = Θ(g(n)).

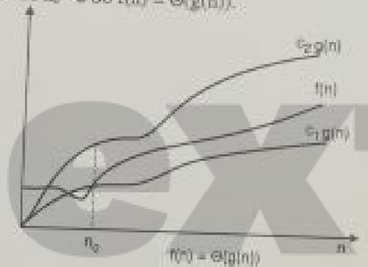


**Fig. P.1.8.3**

### 1.8.4 Little-Oh (o) Notation

- Big-O is used as a tight upper-bound on the growth of an algorithm's effort, even though, as written, it can also be a loose upper-bound. "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.
- We say that f(n) is o(g(n)) if for any real constant c > 0, there exists an integer constant $n_0 \geq 1$ such that f(n) < c * g(n) for every integer n ≥ $n_0$.

  0< = f(n) < c × g(n)

**Example**

$2n$ is $o(n^2)$ as $2n <= n^2$ for all n >=2 and c >=1.

### 1.8.5 Little-Omega (ω) Notation

- We use ω notation to denote a lower bound that is not asymptotically tight.
- We say that f(n) is ω(g(n)) if for any real constant c > 0, there exists an integer constant $n_0 \geq 1$ such that f(n) > c × g(n) ≥ 0 for every integer n ≥ $n_0$.

$f(n) > = c \times g(n)$.

**Example**

$n^2/2$ is $\omega(n)$ as $n^2/2 >= n^2$ for all $n >= 2$ and $c >= 1$.

**Syllabus Topic : Asymptotic Analysis**

## 1.9 Asymptotic Analysis

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation / framing of its run time performance. Using Asymptotic Analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.
- Asymptotic Analysis is input bound that means if there is no input to the algorithm, it is concluded to work in a constant time other than the "input" all other factors are considered constant.
- Asymptotic Analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and maybe for another operation it is $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be linearly the same if n is significantly small.

### 1.9.1 Guidelines for Finding out the Time Complexity of a Piece of Code

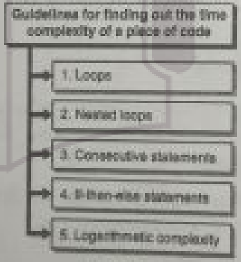

Guidelines for finding out the time complexity of a piece of code
1. Loops
2. Nested loops
3. Consecutive statements
4. If-then-else statements
5. Logarithmetic complexity

Fig. C1.5

**→ 1. Loops**

The running time of loop is the running time of the statements inside the loop multiplied by the number of iterations.

```
# executes n times
For i in range (0, n):
        Print "Current Number:", i              ← Constant time
Therefore, total time = a constant c * n = cn = O(n).
```

**→ 2. Nested loops**

Analyze inside out. Total running time is the product of the size of all the loops.

```
      # outer loop executed n times
      For i in range (0, n):
            #inner loop executes n times
            For j in range (0, n):
Print " i value %d and j value %d" %(i, j)← Constant tim
Therefore, total time = c * n * n
= cn² = O(n²).
```

**→ 3. Consecutive statements**

Add the time complexities of each statements

```
n = 100
#executes n times
fori in range(0,n):
print 'Current Number:', i                    #constant time
#outer loop executed n times
fori in range(0,n):
# inner loop executes n times
for j in range (0,n):
print 'i value %d and j value n/od' %(i,j)     #constant time
Therefore,
Total time = c0 + c1n + c2n² = O(n²).
```

**→ 4. If-then-else statements**

Worst-case running time: the test, plus either thenpart or the else part(whichever is the larger).

```
if n == 10:
print "Wrong Value"                   #constant time
print n
else:
fori in range(0,n):                   #n times
print 'Current Number:', I            #constant time
Therefore,
Total time = c0 + c1 * n = o(n).
```

**➜ 5. Logarithemetic complexity**

An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by 1/2).

```
def Logarithms(n):
i =1
whilei <= n:
i= i * 2
printi
Logarithms(100)
```

If we observe carefully, the value of i is doubling every time. Initially i = 1, in next step i = 2, and in subsequent steps i = 4, 8 and so on.

Let us assume: that the loop is executing some k times.

At k^th^ step $2^k = n$ and we come out of loop.

Taking logarithm on both sides.

```
Log(2^k) = log n
klog2 = logn
k = log2n          //if we assume base-2
Total time = O(log2 n).
```

**Syllabus Topic : Properties of Notations**

## 1.10 Properties of Notations

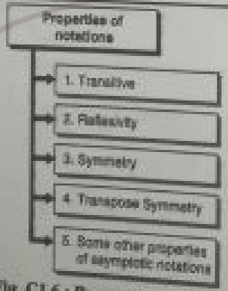


Fig. C1.6 : Properties of Notations

**➜ 1. Transitive**

– If f(n) = Θ(g(n)) and g(n) = Θ(h(n)), then f(n) = Θ(h(n))

– If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n))
– If f(n) = o(g(n)) and g(n) = o(h(n)), then f(n) = o(h(n))
– If f(n) = Ω(g(n)) and g(n) = Ω(h(n)), then f(n) = Ω(h(n))
– If f(n) = ω(g(n)) and g(n) = ω(h(n)), then f(n) = ω(h(n))

**➜ 2. Reflexivity**

– f(n) = Θ(f(n))
– f(n) = O(f(n))
– f(n) = Ω(f(n))

**➜ 3. Symmetry**

f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))

**➜ 4. Transpose Symmetry**

– f(n) = O(g(n)) if and only if g(n) = Ω(f(n))
– f(n) = o(g(n)) if and only if g(n) = ω(f(n))

**➜ 5. Some other properties of asymptotic notations**

– If f (n) is O(h(n)) and g(n) is O(h(n)), then f (n) + g(n) is O(h(n)).
– The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and b ≠ 1.
– $\log_a n$ is O(log n) for any positive a ≠ 1, where log n = $\log_2 n$.

**Syllabus Topic : Commonly Used Logarithms and Summations**

## 1.11 Commonly Used Logarithms and Summations

☞ **Logarithms**

| | |
|---|---|
| $\log x^y = y \log x$ | $\log xy = \log x + \log y$ |
| $\log n = \log^n_{10}$ | $\log^k n = (\log n)^k$ |
| $\log n = \log (\log n)$ | $\log x/y = \log x - \log y$ |
| $a^{\log^x_b} = x^{\log^a_b}$ | $\log^x_b = \log^x_a / \log^b_a$ |

☞ **Arithmetic series**

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

☞ **Geometric series**

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \quad (x \neq 1)$$

**Harmonic series**

$$\sum_{k=1}^{n} \frac{1}{k} 1 + \frac{1}{2} + \ldots\ldots + \frac{1}{n} \approx \log n$$

**Other important formulae**

$$\sum_{k=1}^{n} \log n = n \log n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \ldots\ldots + n^p \approx \frac{1}{p+1} n^{p+1}$$

**Syllabus Topic : Performance Characteristics of Algorithms**

## 1.12 Performance Characteristics of Algorithms

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of :

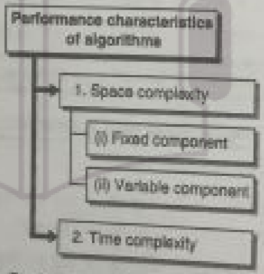


Fig. C1.7 : Performance Characteristics of Algorithms

### 1.12.1 Space Complexity

**Space Complexity** of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space need by any algorithm is the sum of following components.

(i) **Fixed Component**

This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables and constants variables.

(ii) **Variable Component**

This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

- Therefore the total space requirement of any algorithm 'A' can be provided as :

  Space(A) = Fixed Components(A) + Variable Components(A)

- Among both fixed and variable component the variable part is important to be determined accurately, so that the actual space requirement can be identified for an algorithm 'A'. To identify the space complexity of any algorithm following steps can be followed :

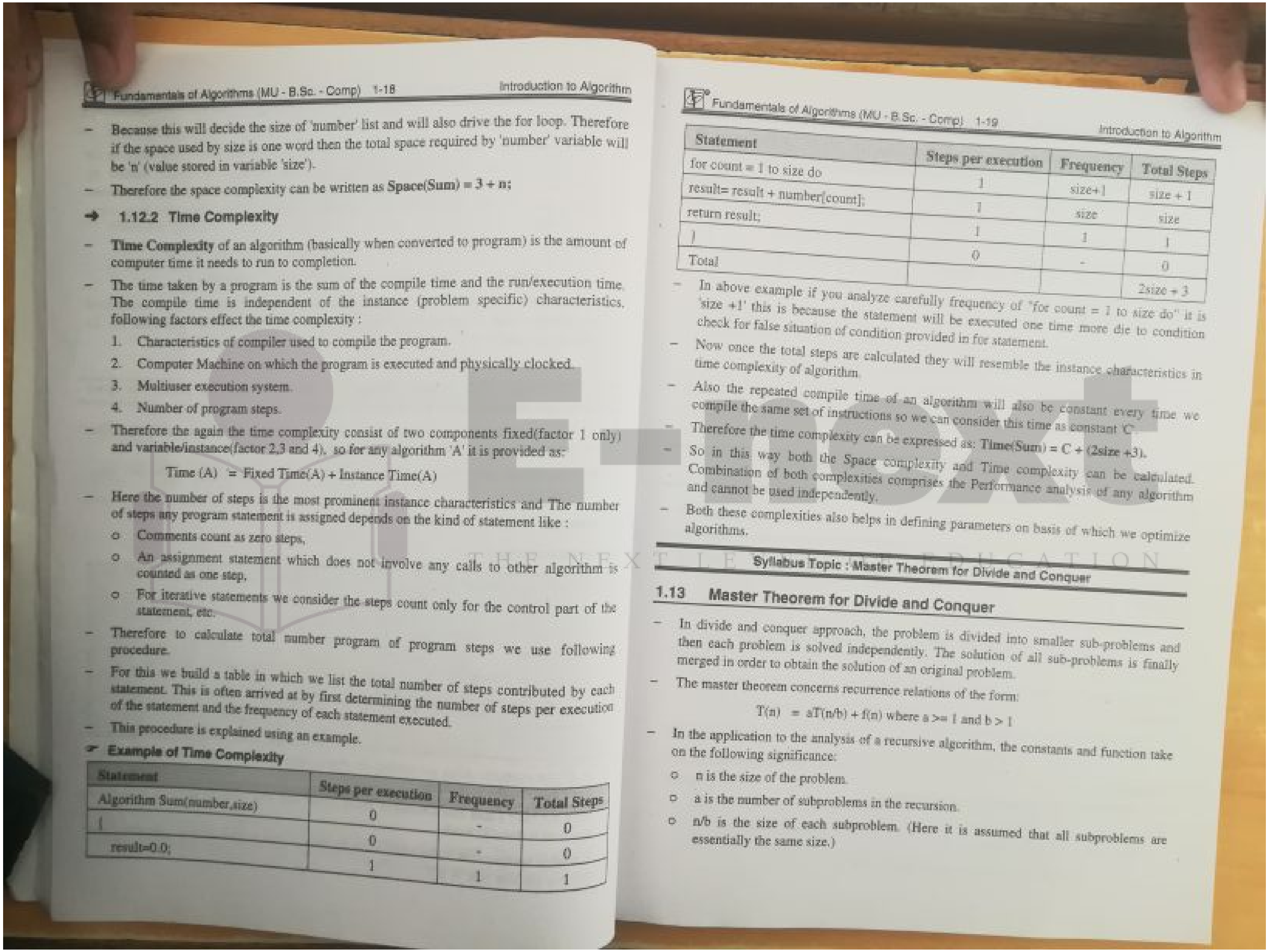
1. Determine the variables which are instantiated by some default values.
2. Determine which instance characteristics should be used to measure the space requirement and this is will be problem specific.
3. Generally the choices are limited to quantities related to the number and magnitudes of the inputs to and outputs from the algorithms.
4. Sometimes more complex measures of the interrelationships among the data items can used.

**Example of Space Complexity**

**Space Complexity**

```
Algorithm Sum(number, size)                    \\ procedure will produce sum of all numbers
                                               \\provided in 'number' list
{
    result=0.0;
    for count = 1 to size do                   \\will repeat from 1,2,3,4,....size times
        result= result + number[count];
    return result;
}
```

- In above example, when calculating the space complexity we will be looking for both fixed and variable components.
- Here we have **Fixed components** as 'result','count' and 'size' variable there for total space required is three(3) words.
- **Variable components** is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n').



- Because this will decide the size of 'number' list and will also drive the for loop. Therefore if the space used by size is one word then the total space required by 'number' variable will be 'n' (value stored in variable 'size').
- Therefore the space complexity can be written as **Space(Sum) = 3 + n;**

### 1.12.2 Time Complexity

- **Time Complexity** of an algorithm (basically when converted to program) is the amount of computer time it needs to run to completion.
- The time taken by a program is the sum of the compile time and the run/execution time. The compile time is independent of the instance (problem specific) characteristics. following factors effect the time complexity :
  1. Characteristics of compiler used to compile the program.
  2. Computer Machine on which the program is executed and physically clocked.
  3. Multiuser execution system.
  4. Number of program steps.
- Therefore the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 and 4). so for any algorithm 'A' it is provided as:

  Time (A) = Fixed Time(A) + Instance Time(A)

- Here the number of steps is the most prominent instance characteristics and The number of steps any program statement is assigned depends on the kind of statement like :
  - Comments count as zero steps,
  - An assignment statement which does not involve any calls to other algorithm is counted as one step,
  - For iterative statements we consider the steps count only for the control part of the statement, etc.
- Therefore to calculate total number program of program steps we use following procedure.
- For this we build a table in which we list the total number of steps contributed by each statement. This is often arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed.
- This procedure is explained using an example.

**Example of Time Complexity**

| Statement | Steps per execution | Frequency | Total Steps |
|---|---|---|---|
| Algorithm Sum(number,size) | 0 | - | 0 |
| { | 0 | - | 0 |
| result=0.0; | 1 | 1 | 1 |
| for count = 1 to size do | 1 | size+1 | size + 1 |
| result= result + number[count]; | 1 | size | size |
| return result; | 1 | 1 | 1 |
| } | 0 | - | 0 |
| Total | | | 2size + 3 |

- In above example if you analyze carefully frequency of "for count = 1 to size do" it is 'size +1' this is because the statement will be executed one time more die to condition check for false situation of condition provided in for statement.
- Now once the total steps are calculated they will resemble the instance characteristics in time complexity of algorithm.
- Also the repeated compile time of an algorithm will also be constant every time we compile the same set of instructions so we can consider this time as constant 'C'.
- Therefore the time complexity can be expressed as: **Time(Sum) = C + (2size +3).**
- So in this way both the Space complexity and Time complexity can be calculated. Combination of both complexities comprises the Performance analysis of any algorithm and cannot be used independently.
- Both these complexities also helps in defining parameters on basis of which we optimize algorithms.

**Syllabus Topic : Master Theorem for Divide and Conquer**

## 1.13 Master Theorem for Divide and Conquer

- In divide and conquer approach, the problem is divided into smaller sub-problems and then each problem is solved independently. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.
- The master theorem concerns recurrence relations of the form:

  $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

- In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:
  - n is the size of the problem.
  - a is the number of subproblems in the recursion.
  - n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)

- o $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.
- For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form then we can directly give the answer without fully solving it:
- If the recurrence is of the form,

$$T(n) = aT(n/b) + \Theta(n^k \log^p n), \text{ where } a >= 1,\ b > 1,\ k >= 0,$$

and p is a real number, then:

1. If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$.
2. If $a = b^k$
   a) If $p > -1$ then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) If $p = -1$ then $T(n) = \Theta(n^{\log_b a} \log \log n)$
   c) If $p < -1$ then $T(n) = \Theta(n^{\log_b a})$
3. If $a < b^k$
   a) If $p >= 0$ then $T(n) = \Theta(n^k \log^p n)$
   b) If $p < 0$ then $T(n) = O(n^k)$

**Syllabus Topic : Master Theorem for Divide and Conquer Problems and Solutions**

## 1.13.1 Master Theorem for Divide and Conquer : Problems and Solutions

**Example 1.13.1**

Find Recurrence for binary search : $T(n) = T(n/2) + 1$.

**Solution :**

Here, $a = 1$, $b = 2$, $k = 0$, Hence $a = b^k$

By case 2 of Master theorem, $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$

**Example 1.13.2**

Find Recurrence for merge sort: $T(n) = 2 \cdot T(n/2) + 4n$

**Solution :**

Here, $a = 2$, $b = 2$, $k = 1$, Hence $a = b^k$

By Case 2 of Master theorem, $T(n) = \Theta(n \cdot \log n)$

**Example 1.13.3**

Find Recurrence for binary search : $T(n) = 4\,T(n/2) + n$

**Solution :**

Here, $a = 4$, $b = 2$, $k = 1$, Hence $a > b^k$

By Case 1 of Master theorem, $T(n) = \Theta(n^2)$

**Example 1.13.4**

Find Recurrence for binary search : $T(n) = 4\,T(n/2) + n^2$

**Solution :**

Here, $a = 4$, $b = 2$, $k = 2$, Hence $a = b^k$

By Case 2(a) of Master theorem, $T(n) = \Theta(n^2 \log n)$

**Example 1.13.5**

Find Recurrence for binary search : $T(n) = 4\,T(n/2) + n^3$

**Solution :**

Here, $a = 4$, $b = 2$, $k = 3$, Hence $a < b^k$

By Case 3(a) of Master theorem, $T(n) = \Theta(n^3)$

**Example 1.13.6**

Find Recurrence for binary search : $T(n) = T(n/2) + n^2$

**Solution :**

Here, $a = 1$, $b = 2$, $k = 2$, Hence $a < b^k$

By Case 3(a) of Master theorem, $T(n) = \Theta(n^2)$

**Example 1.13.7**

Find Recurrence for binary search : $T(n) = 2^n\,T(n/2) + n^n$

**Solution :**

Does not apply ($a$ is not constant)

**Example 1.13.8**

Find Recurrence for binary search : $T(n) = 2T(n/2) + n \log n$

**Solution :**

Here, $a = 2$, $b = 2$, $k = 1$, Hence $a = b^k$

By Case 2(a) of Master theorem, $T(n) = \Theta(n \log^2 n)$

**Example 1.13.9**

Find Recurrence for binary search : $T(n) = 2T(n/2) + n/\log n$

**Solution :**

Here, $a = 2$, $b = 2$, $k = 1$, Hence $a = b^k$

By Case 2(b) of Master theorem,

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

**Example 1.13.10**

Find Recurrence for binary search : $T(n) = 2T(n/4) + n^{0.51}$

**Solution :** Here, $a = 2$, $b = 4$, $k = 0.51$, Hence $a < b^k$

By Case 3(b) of Master theorem, $T(n) = \Theta(n^{0.51})$

**Example 1.13.11**

Find Recurrence for binary search : $T(n) = 0.5T(n/2) + 1/n$

**Solution :**

Does not apply ($a < 1$)

**Example 1.13.12**

Find Recurrence for binary search : $T(n) = 64T(n/8) - n^2 \log n$

**Solution :**

Does not apply (function is not positive)

**Example 1.13.13**

Find Recurrence for binary search : $T(n) = 3T(n/3) + n/2$

**Solution :**

Here, $a = 3$, $b = 3$, $k = 1$, Hence $a = b^k$

By Case 2(a) of Master theorem, $T(n) = \Theta(n \log n)$

**Example 1.13.14**

Find Recurrence for binary search : $T(n) = 3T(n/3) + \sqrt{n}$.

**Solution : :**

Here, $a = 3$, $b = 3$, $k = 1$, Hence $a = b^k$

By Case 1 of Master theorem, $T(n) = \Theta(n)$

**Example 1.13.15**

Find Recurrence for binary search : $T(n) = 16T(n/4) + n!$

**Solution :**

Here, $a = 16$, $b = 4$, $k = 1$, Hence $a > b^k$

By Case 3(a) of Master theorem, $T(n) = \Theta(n!)$

**Example 1.13.16**

Find Recurrence for binary search : $T(n) = T(n/2) + 4$

**Solution :**

Here, $a = 1$, $b = 2$, $k = 0$, Hence $a = b^k$

By Case 2 of Master theorem, $T(n) = \Theta(\log n)$

**Syllabus Topic : Master Theorem for Subtract and Conquer Recurrences**

## 1.14 Master Theorem for Subtract and Conquer Recurrences

Let T(n) be a function defined on positive n as shown below :

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1 \end{cases}$$

for some constants c, $a>0$, $b>0$, $k>=0$and function f(n).

If f(n) is $O(n^k)$, then

1. If $a<1$ then $T(n) = O(n^k)$
2. If $a=1$ then $T(n) = O(n^{k+1})$
3. if $a>1$ then $T(n) = O(n^k a^{n/b})$

Let's try some example :

**Example 1.14.1**

$T(n) = 3T(n-1)$ when $a>0$

$= 1$ otherwise

**Solution :**

Here $a = 3$, $b = 1$ and $d = 0$ hence $T(n) = O(n^0 3^{n/1})$ $(a > 1)$

that means $T(n) = O(3^n)$.

**Example 1.14.2**

$T(n) = 5T(n-3)+O(n^2)$ when $n>0$

$= 1$ otherwise

**Solution :**

Here $a = 5$, $b = 3$, $d = 2$ hence $T(n) = O(n^2 5^{n/3})$.

**Example 1.14.3**

$T(n) = 2T(n-1) - 1$, when $n>0$

$= 1$, when $n<=0$

**Solution :**

This recurrence can't be solved using above method since function is not of form

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

**Syllabus Topic : Method of Guessing and Confirming**

## 1.15 Method of Guessing and Confirming

- Let us discuss about a method which can be used to solve any recurrence. The basic idea behind this method is, guess the answer; and then prove it correct by induction.
- In other words, it addresses the question: What if the given recurrence doesn't seem to match with any of these (master theorems) methods?
- If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).
- As an example, consider the recurrence $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorems.
- Carefully observing the recurrence gives us the impression that it is similar to divide and conquer method (diving the problem into $\sqrt{n}$ sub-problems each with size $\sqrt{n}$). As we can see, the size of the sub-problems at the first level of recursion is n.
- So, let us guess that $T(n) = O(n \log n)$, and then try to prove that our guess is correct.
- Let's start by trying to prove an upper bound $T(n) \le c\, n \log n$ :

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\le \sqrt{n}\, c\sqrt{n} \log \sqrt{n} + n \\ &= n.\, c \log \sqrt{n} + n \\ &= n.c.1/2.\log n + n \\ &\le c\, n \log n \end{aligned}$$

- The last inequality assumes only that $1 \le c.1/2.\log n$. This is correct if n is sufficiently large and for any constant c, no matter how small. From the above proof, we can see that our guess is correct for upper bound. Now, let us prove the lower bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\ge \sqrt{n}\, k \sqrt{n} \log \sqrt{n} + n \end{aligned}$$

$$\begin{aligned} &= n.\, k \log \sqrt{n} + n \\ &= n.k.1/2.\log n + n \\ &\ge k\, n \log n \end{aligned}$$

- The last inequality assumes only that $1 \ge k.1/2.\log n$. This is incorrect if n is sufficiently large and for any constant k. From the above proof, we can see that our guess is incorrect for lower bound.
- From the above discussion, we understood that $\Theta(n \log n)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly :

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + n \ge n$$

Now, let us prove the upper bound for this $\Theta(n)$.

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\le \sqrt{n}\,.c.\,\sqrt{n} + n \\ &= n.\,c + n \\ &= n\,(c + 1) \\ &\le c\,n \end{aligned}$$

From the above induction, we understood that $\Theta(n)$ is too small and $\Theta(n \log n)$ is too big. So, we need something bigger than n and smaller than n log n? How about $n\sqrt{\log n}$?

Proving upper bound for $n\sqrt{\log n}$:

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\le \sqrt{n}\,.c.\,\sqrt{n}\left(\sqrt{\log \sqrt{n}}\right) + n \\ &= n.\, c.1/\sqrt{2} \log \sqrt{n} + n \\ &\le c\, n\log\sqrt{n} \end{aligned}$$

Proving lower bound for $n\sqrt{\log n}$ :

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\ge \sqrt{n}.\,k.\,\sqrt{n}\left(\sqrt{\log \sqrt{n}}\right) + n \\ &= n.\,k.\,1/\sqrt{2} \log\sqrt{n} + n \\ &\ngeq k\, n \log \sqrt{n} \end{aligned}$$

The last step doesn't work. So, $\Theta(n\sqrt{\log n})$ doesn't work. What else is between n and n log n? How about n log log n?

Proving upper bound for n log log n :

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\le \sqrt{n}.c.\sqrt{n}\sqrt{n}\ \log\log\sqrt{n} + \sqrt{n} + n \\ &= n.\ c.\log\log n - c.\ n + n \\ &\le c\ n\log\log n,\ \text{if } c \ge 1 \end{aligned}$$

Proving lower bound for n log log n:

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\ge \sqrt{n}.\ k.\ \log\log\sqrt{n} + n \\ &= n.\ k.\log\log n - k.\ n + n \\ &\ge k\ n\log\log n,\ \text{if } k \le 1 \end{aligned}$$

From the above proofs, we can see that $T(n) \le c\ n \log\log n$, if $c \ge 1$ and $T(n) \ge k\ n \log\log n$, if $k \le 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $T(n) = \Theta(n \log\log n)$.

**Review Questions**

Q. 1 What is algorithm ? **(Refer section 1.1.1)**

Q. 2 List and explain characteristics of an algorithm. **(Refer section 1.1.2)**

Q. 3 List and explain various asymptotic notation **(Refer section 1.8).**

Q. 4 Explain different performance characteristics of algorithm. **(Refer sections 1.12, 1.12.1 and 1.12.2)**

❑❑❑

UNIT II

# CHAPTER 2 Tree Algorithms

**Syllabus**

What is a Tree? Glossary, Binary Trees, Types of Binary Trees, Properties of Binary Trees, Binary Tree Traversals, Generic Trees (N-ary Trees), Threaded Binary Tree Traversals, Expression Trees, Binary Search Trees (BSTs), Balanced Binary Search Trees, AVL (Adelson-Velskii and Landis) Trees.

**Syllabus Topic : What is a Tree ?**

## 2.1 Tree

– In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order.

– Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows.

– "Tree is a non-linear data structure which organizes data in hierarchical structure."

– Tree represents the nodes connected by edges.

**Syllabus Topic : Glossary**

### 2.1.1 Glossary of Tree

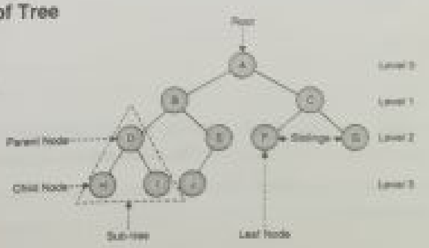


Fig. 2.1.1