Advanced Data Structures and Analysis
ADSA
ITDO 5014

Lakshmi M. Gadhikar

Information Technology Department
Fr. C.R.I.T. , Vashi, Navi Mumbai.

Advanced Data Structures and Analysis ITDO 5014 – ADSA

Course Coverage (13 Weeks):

Module 1 : Introduction	**(1.5 Weeks)**
Module 2 : Advanced Data Structures	**(2 Weeks)**
Module 3 : Divide and Conquer & Greedy Algorithms	**(2.5 Weeks)**
Module 4 : Dynamic Algorithms	**(02 Weeks)**
Module 5 : String Matching	**(2.5 Weeks)**
Module 6 : Advanced Algorithms and NP Problems	**(1.5 Weeks)**
Revision	**(01 Week)**

ITDO 5014 ADSA (39 Hrs)

Module 1 : Introduction (4 Hrs)

Fundamentals of the analysis of algorithms :

- Time and Space Complexity ,
- Asymptotic analysis and notation,
- average and worst case analysis

Recurrences:

- The substitution Method
- Recursive tree Method
- Masters method

Self-learning Topics: Analysis of Time and space complexity of iterative and recursive algorithms

Module 2 : Advanced Data Structures : (5 Hrs)

- B/B+ tree
- Red-Black Trees
- Heap operations
- Implementation of queue using heap
- Topological sort

Self-learning Topics:

Implementation of Red-Black Tree and Heaps.

ITDO 5014 ADSA SYLLABUS

Module 3 : Divide and Conquer AND Greedy Algorithms : (9 Hrs)

1. Introduction to Divide and Conquer
   Analysis of :
2. Binary search
3. Merge sort and Quick sort
4. Finding the minimum and maximum algorithm

3. Introduction to Greedy Algorithms
4. Knapsack problem
5. Job sequencing with deadlines
6. Optimal storage on tape
7. Optimal merge pattern
8. Analysis of All these algorithms and problem solving.

Self-learning Topics:
Implementation of minimum and maximum algorithm, Knapsack problem, Job sequencing using deadlines.

ITDO 5014 ADSA SYLLABUS

Module 4 : Dynamic Algorithms (6 Hrs)

1. Introduction Dynamic algorithms
2. All pair shortest path
3. 0/1 knapsack
4. Travelling salesman problem
5. Matrix Chain Multiplication
6. Optimal binary search tree (OBST)
7. Analysis of All algorithms and problem solving.

Self-learning Topics: Implementation of All pair shortest path, 0/1 Knapsack and OBST.

Module 5 : String Matching (7 Hrs)

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

   Analysis of All algorithms and problem solving.

Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.

ITDO 5014 ADSA SYLLABUS

Module 6 : Advanced Algorithms and NP problems : (6 Hrs)

1. Optimization Algorithms: Genetic algorithm (GA),
2. Approximation Algorithms: Vertex-cover problem,
3. Parallel Computing Algorithms: Fast Fourier Transform,
4. Introduction to NP-Hard and NP-Complete Problems

Self-learning Topics: Implementation of Genetic algorithm and Vertex-cover problem.

Advanced Data Structures & Analysis
ITDO 5014
Course Objectives

The learners will try :

1. **To learn mathematical background for analysis of algorithm**
2. **To learn various advanced data structures.**
3. **To understand the different design approaches of algorithm.**
4. **To solve problems using various strategies such as dynamic programming and greedy method.**
5. **To understand the concept of pattern matching**
6. **To learn advanced algorithms.**

Advanced Data Structures & Analysis
ITDO 5014
Course Outcomes SH 22

After successful completion of the course, the learners will be able to :

CO-ID	CO-Statement
ITDO 5014.1	Recall mathematical aspects and fundamentals of AOA. (BL1 PO1:2 M1) Learner will be able to Identify, analyze and solve
ITDO 5014.2	Demonstrate appropriate advanced data structures.(BL2 PO1:3 PO2:2, M2)
ITDO 5014.3	Demonstrate appropriate algorithmic design techniques and Advanced algorithms. (BL2, PO1:3 M3, M4, M5, M6)
ITDO 5014.4	Apply appropriate algorithmic design technique. (BL3 PO1:3 PO2:3 PO3:3 PO5:3 M3, M4, M5, M6)
ITDO 5014.5	Analyze complexity of different algorithms. (BL4 PO1:3 PO2:3 M3,M4,M5,M6)

Advanced Data Structures & Analysis
ITDO 5014
Course Outcomes

After successful completion of the course, the learners will be able to :

CO-ID	CO-Statement
ITDO 5014.1	Recall mathematical aspects and fundamentals of AOA. (BL1 PO1:2 M1)
ITDO 5014.2	Demonstrate appropriate advanced data structures.(BL2 PO1:3 PO2:2, M2)
ITDO 5014.3	Demonstrate appropriate algorithmic design techniques and Advanced algorithms. (BL2, PO1:3 M3, M4, M5, M6)
ITDO 5014.4	Apply appropriate algorithmic design technique. (BL3 PO1:3 PO2:3 PO3:3 PO5:3 M3, M4, M5, M6)
ITDO 5014.5	Analyze complexity of different algorithms. (BL4 PO1:3 PO2:3 M3,M4,M5,M6)

Advanced Data Structures & Analysis ITDO 5014

Course Outcomes

CO-ID	CO-Statement	Tool 1	Tool 2
ITDO 5014.1	Recall mathematical aspects and fundamentals of AOA.	Assignment Test-1	Internal Assessment-1
ITDO 5014.2	Demonstrate appropriate advanced data structures.	Assignment Test-1	Internal Assessment-1
ITDO 5014.3	Demonstrate appropriate algorithmic design techniques and Advanced algorithms.	Internal Assessment-1	Internal Assessment-2
ITDO 5014.4	Apply appropriate algorithmic design technique.	Assignment Test-2	Internal Assessment-2
ITDO 5014.5	Analyze complexity of different algorithms.	Internal Assessment-1	Internal Assessment-2

Advanced Data Structures & Analysis
ITDO 5014

Text Books:

T1. Introduction to ALGORITHMS, Cormen, Leiserson, Rivest, Stein, PHI. (E-Book)

T2. Algorithms: Design and Analysis, Harsh Bhasin, OXFORD.

T3. Fundamentals of Computer Algorithms, Horowitz, Sahani, Rajsekaran, Universities Press. (E-Book)

T4. C and Data structures, Deshpande, Kakde, Dreamtech Press.

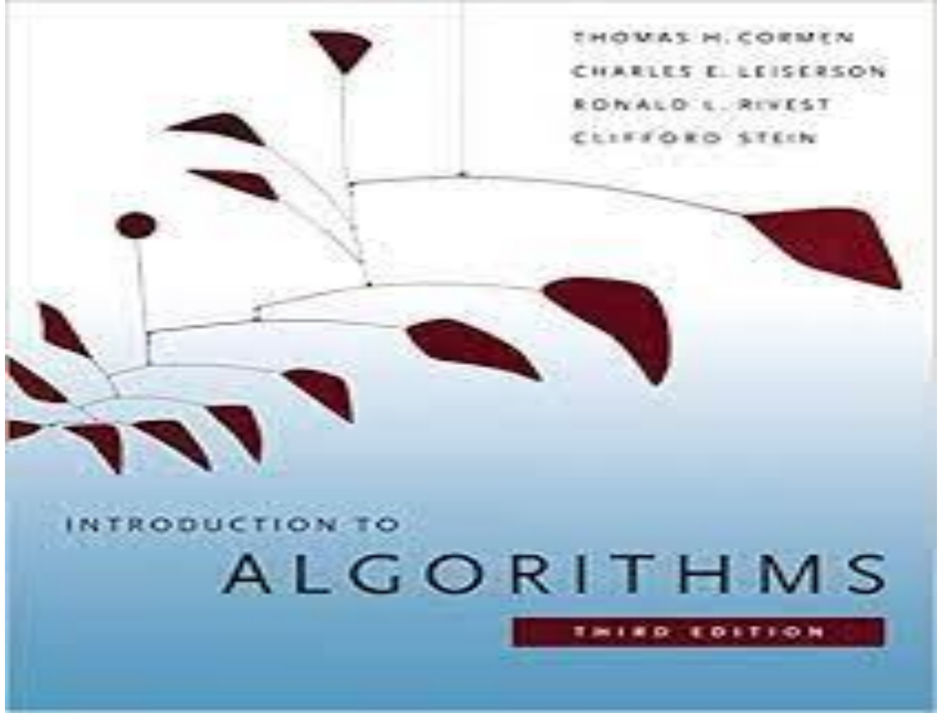
Reference Books:

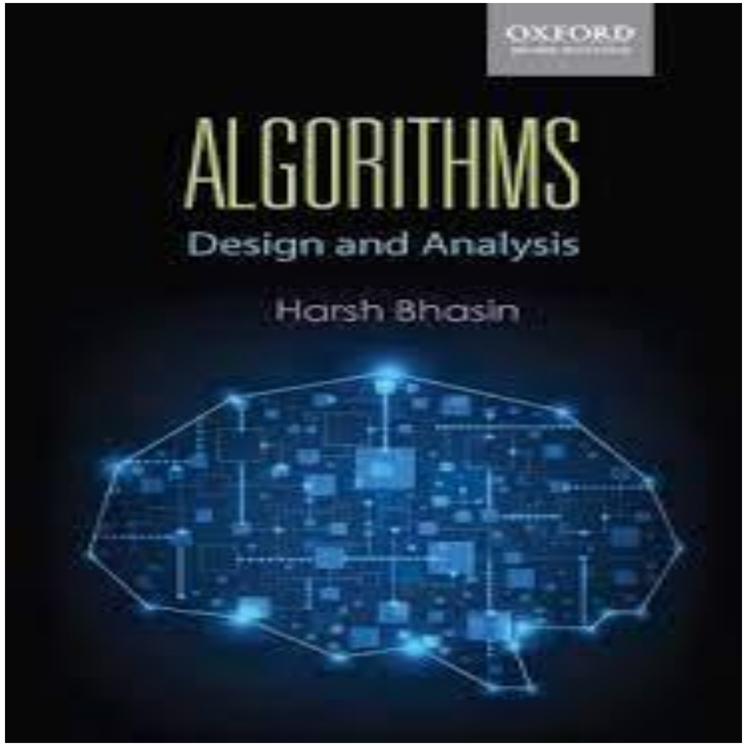
R1. Data Structures and Algorithms in C++, Goodritch, Tamassia, Mount, WILEY. (String Matching) (E-Book)

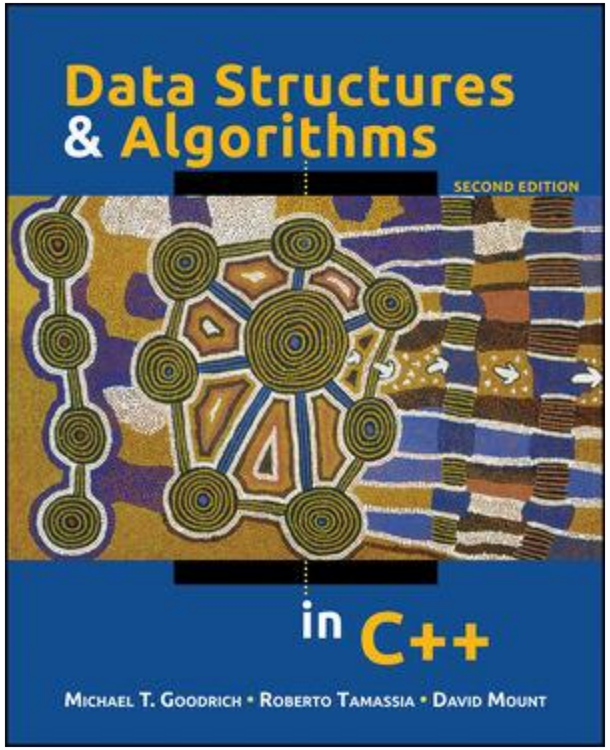
R2. Data Structures using C, Reema Thareja, OXFORD.

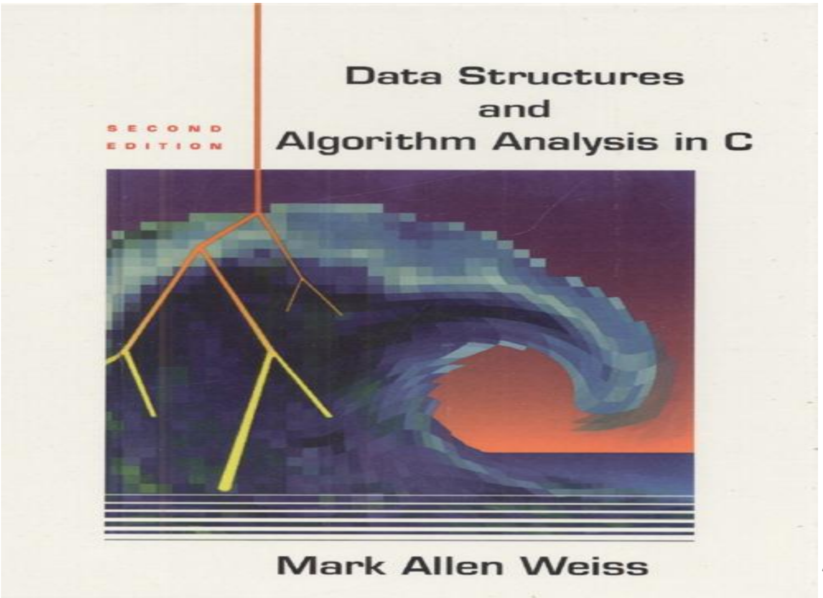
R3. Data Structures and Algorithm Analysis in C, Mark A. Weiss, Pearson.

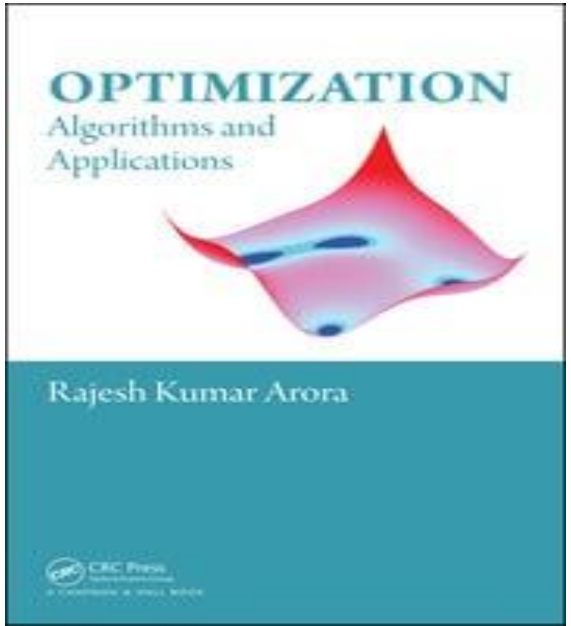
R4. Optimization Algorithms and Applications, By Rajesh Kumar Arora, Chapman and Hall. (E-Book)


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION


OXFORD
ALGORITHMS
Design and Analysis
Harsh Bhasin


Data Structures
& Algorithms
SECOND EDITION
in C++
MICHAEL T. GOODRICH • ROBERTO TAMASSIA • DAVID MOUNT


SECOND EDITION
Data Structures
and
Algorithm Analysis in C
Mark Allen Weiss


OPTIMIZATION
Algorithms and
Applications
Rajesh Kumar Arora
CRC Press

Useful Links for Self learning :

1. https://nptel.ac.in/courses/106/106/106106131/
2. https://swayam.gov.in/nd1_noc19_cs47/preview
3. https://www.coursera.org/specializations/algorithms
4. https://www.mooc-list.com/tags/algorithms

Examination for the course ITDO 5014 – ADSAOA :

Internal Assessment Test 1 (20 marks)
Internal Assessment Test 2 (20 marks)

Assignment Tests 1 (20 marks)
Assignment Tests 2 (20 marks)

Approximately 40% to 50% of syllabus content must be covered in Test 1 and Remaining 40% to 50% of syllabus contents must be covered in Test2.

Open book / home Assignments (20 marks)
Quizzes

End Semester Examination for 80 marks :

On complete syllabus.

End Semester Examination for 80 marks :

Question paper pattern :

Weightage of each module in end semester examination is expected to be/will be proportional to number of respective lecture hours mentioned in the syllabus.

Question paper will comprise of total six questions, each carrying 20 marks.

Q.1 will be compulsory and should cover maximum contents of the syllabus.

Remaining question will be mixed in nature (for example if Q.2 has part (a) from module 3, then part (b) will be from any other module. (Randomly selected from all the modules.)
Total four questions need to be solved.

Advanced Data Structures and Analysis
ITDO 5014

Module 1
Introduction

Lakshmi M. Gadhikar
Information Technology Department
Fr. C.R.I.T. , Vashi, Navi Mumbai.

Advanced Data Structures and Analysis
ITDO 5014

Module 1 : Introduction

Fundamentals of the analysis of algorithms :

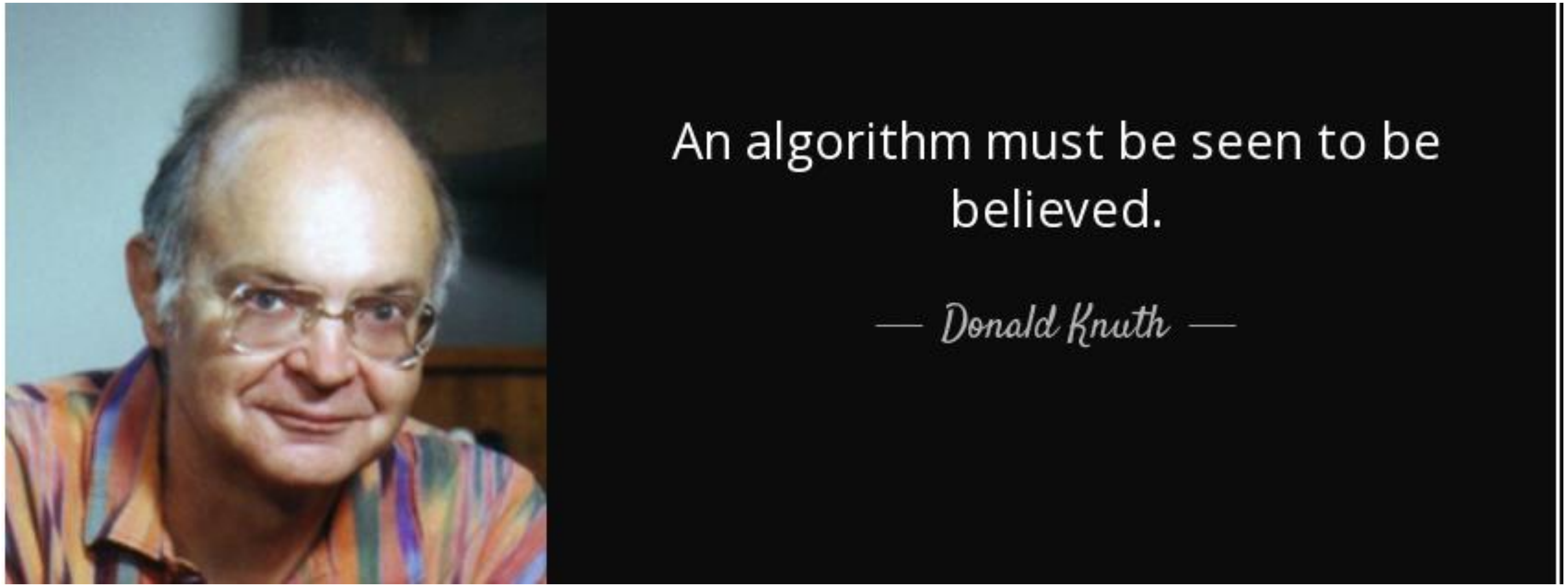
- **Time and Space Complexity ,**
- **Asymptotic analysis and notation,**
- **average and worst case analysis**

Recurrences:

- **The substitution Method**
- **Recursive tree Method**
- **Masters method**

Self-learning Topics: Analysis of Time and space complexity of iterative and recursive algorithms

Advanced Data Structures and Analysis
ITDO 5014



By 2025, 80 percent of the functions doctors do will be done much better and much more cheaply by machines and machine learned algorithms.

- Vinod Khosla

Questions :

Q1. What is an Algorithm ?

Q2. Why study Algorithms ?

Q2. Why Analyze Algorithms ?

Fundamentals of the analysis of algorithms :

Q1. What is an Algorithm ?

Problem Solving

Program P in C, C++, java , Python etc.

Algorithms : A1 A2 A3 A4 ……………………..

In order to solve the problem , we write a program.

A single program may be written in multiple ways – multiple algorithms. Eg. sort

Each algo. may take different amount of

i. Time for execution and

ii. Memory space for storage

Out of the several options, whichever algorithm takes minimum time and space ,We implement that algorithm as a program in chosen language.

Q2. Why study Algorithms ?

Algorithms are used in all domains of Computer and Information Technology such as :

- **High Performance Computing**
- **AI and Deep learning**
- **Data Science**
- **Networking and security**
- **Internet and Social Networking etc.**

Algorithms also find their applications in other disciplines and optimization problems such as :

- **Travelling Salesman problem**
- **Flow shop scheduling Job sequencing with deadlines etc.**

Advanced Data Management Technologies (ADMT) :

Query Processing and Optimization:
Transactions Management and Concurrency:
Advanced Data Management techniques
Distributed Databases
Data Warehousing,
Multi Dimensional Modeling and OLAP
ETL Process

Algorithm :

- An algorithm is any well defined computational procedure (i.e. a precise set of steps) that
  takes some value , or set of values as input and
  produces some value or a set of values as output.

```
Fibonacci(n)
{
  if (n==0) return 0;
  else if (n==1) return 1;
  else return ( Fibonacci(n-1)
                + Fibonacci(n-2);
}
```

- An algorithm may be viewed as a tool for solving a well specified computational problem.
  Eg. Binary , Seq. Search, Sort: Bubble, quick etc.
  TSP, APSP, Matrix Chain Multiplication etc.

- An algorithm is a method of solving problems

```
Fibo(n)
{
    Int A[n];
    A[0] = 0;
    A[1] = 1;
    For (i =  2 ; i<= n ; i++ )
        A[i] = A[i-1] +A[i-2];
    Return A[n];
}
```

n	0	1	2	3	4	5	6	7	8	9
Fibo(n)	0	1	1	2	3	5	8	13	21	34

Example of an Algorithm :

An algorithm to compute average of N numbers :

```
sum = 0.0;
For  I =0; I < N; I++
        sum = sum + a[I]
Average = sum / N
```

Difference between an algorithm and program :

Design phase : Algorithm

Implementation of an algorithm : Program

Program : Concrete expression of an algorithm in a particular programming language is called as a program.

Problem -> Multiple Algorithms -> Multiple programs

Properties of the algorithm :

1. **Finiteness :**

   An algorithm must always terminate after a finite number of steps.

2. **Definiteness / Precision :**

   Each step of an algorithm must be precisely defined;
   i.e. the actions to be carried out must be rigorously and unambiguously specified for each case.

3. **Input :**

   An algorithm has zero or more inputs, i.e, quantities which are given to it initially
   -before the algorithm begins.

4. **Output :**

   An algorithm has one or more outputs i.e, quantities which have a specified relation to the inputs.

5. **Effectiveness :** An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

6. **Generality** – the algorithm applies to a set of inputs.

An algorithm is said to be correct if,

for every input instance (possible set of i/ps) ,

it halts with the correct output.

---

1. Finiteness :

```
i= 0
While i< 10 do
{   -----
    i--;
}
```

2. Definiteness : do while (i>0 and i< 10) // definite do while (i<0 and i> 10) // indefinite

5. Effectively : In order to solve a problem u call 5 functions with no guarantee that whether each of them will execute correctly and each of them will terminate in finite amt of time or not

Ambiguous Algorithm

- Take two pieces of bread
- Put peanut butter on one side of one piece
- Put jelly on one side of the other piece
- Put the pieces together
- Output of this algorithm is. . . .

A Messy Sandwich

- Why?
- The algorithm did not specify that the pieces of bread should be put together so that the peanut butter and jelly are on the inside
- Computers do ***exactly*** what they're told or, worse, something ***undefined***
- Precision is important

Q2. Why study Algorithms ?

Algorithms are used in all domains of Computer and Information Technology such as :

High Performance Computing
AI and Deep learning
Data Science
Networking and security
Internet and Social Networking etc.

Algorithms also find their applications in other disciplines and optimization problems such as :

Travelling Salesman problem
Flow shop scheduling etc.

Complexity of Algorithm:

It is very convenient to classify algorithms based on :

the relative amount of time or

relative amount of space they require and

specify the growth of time /space requirements as a function of the input size.

Thus, we have the notions of:

1. Time Complexity:

Running time of the program as a function of the size of input T(n)

n = input size

2. Space Complexity:

Amount of computer memory required during the program execution, as a function of the input size. S(n)

Introduction / Fundamentals of the analysis of algorithms :

Problem Solving

↓

Program P in C, C++, java , Python etc

Algorithms : A1 A2 A3 A4 ……………………..

Out of the several options, whichever algorithm takes minimum time and space ,We implement that algorithm as a program in chosen language.

Analysis of Algorithms :

We analyze a set of possible algorithms to find the one that takes the LEAST TIME FOR EXECUTION (OR LEAST MEMORY SPACE FOR STORAGE or both) called as the BEST CASE and implement that algorithm as a program in chosen language.

Q3. Why Analyze Algorithms ?

We analyze algorithms to find their efficiency **in terms of**

Time complexity and

Space complexity

And

Choose the one with the best / average case **running time and/or storage space requirements.**

Best Case Running Time :

The term *best-case performance* is used to describe an algorithm's behavior under optimal conditions.
Best case gives the minimum time required for execution of the algorithm.

For example, the best case for a simple linear search on a list occurs when the desired element is the first element of the list.

T(1) : One comparison required to get ans.

n = 7

0	1	2	3	4	5	6
10	4	2	5	9	3	8

Worst Case Running Time :

The term wors*t-case performance* is used to describe an algorithm's behavior under worst possible case of input instance i.e. under worst conditions.

Worst case gives the maximum time required for execution of the algorithm.

The worst case running time of an algorithm is an upper bound on the running time of an algorithm for any input.

n = 7

0	1	2	3	4	5	6
10	4	2	5	9	3	8

For example, the worst case for a simple linear search on a list occurs when the desired element is the LAST element of the list. T(n) : n comparisons required

Knowing it gives us a guarantee that the algorithm will never take any longer. There is no need to make an educated guess about the running time.

Average case running Time :

The expected behavior when the input is randomly drawn from a given distribution.

The average-case running time of an algorithm is an estimate of the running time for an "average" input.

Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences.

Often it is assumed that all inputs of a given size are equally likely.

n = 7

0	1	2	3	4	5	6
10	4	2	5	9	3	8

[Ref: Cormen - pg 44]

Development and choice of algorithms is rarely based on best-case performance:

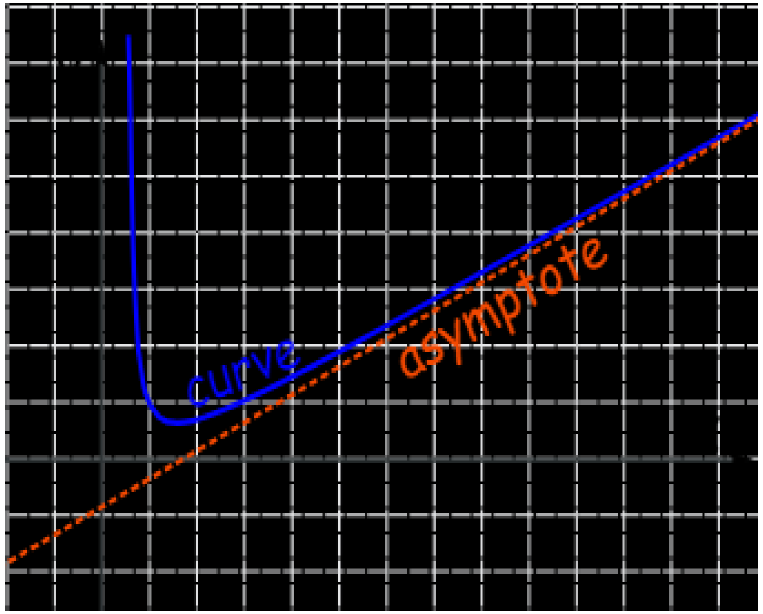
Most academic and commercial enterprises are more interested in :

Improving Average-case complexity and worst-case performance.

Q1. Explain Asymptotic notations with the help of graph (10M)

Q2. Explain O, Ω , Θ with the help of graph and represent the following functions using above notations. (10M)

Asymptotic notations used commonly in performance analysis to represent the complexity of algorithms are :



1. Big Oh (O = Upper Bound = Worst case time)
2. Big Omega (Ω = Lower Bound = Best case time)
3. Big Theta (Θ = Average case time)

1. Big Oh = Upper Bound = Worst case running time of an algorithm :

Big Oh notation is used to give the maximum time required for execution of the algorithm.

i.e. it is an upper bound on the running time of an algorithm for any input.

$f(n)$ is said to be big Oh of g of n written as $f(n) = O(g(n))$

Eg. $F(n) = 3n + 2$, $g(n) = n$

iff there exist a positive real constant C and a positive integer n_0 such that

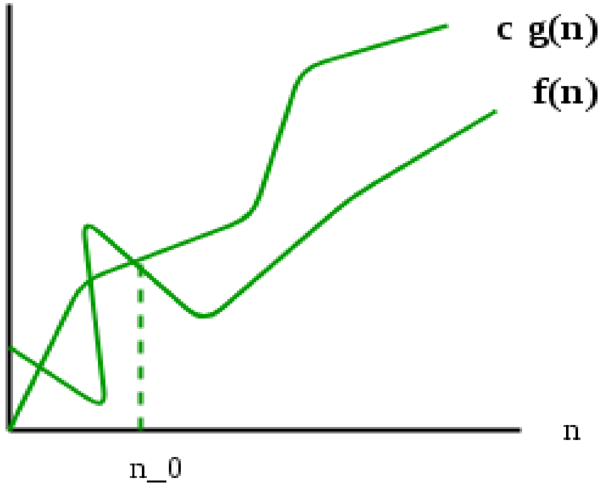
$0 <= f(n) <= Cg(n)$ For all $n >= n_0$

$\Rightarrow$ We can bound the function f(n) by another function Cg(n) , such that after some value n0 , the value of f(n) is always less than Cg(n).

For n >= n0

C > 0

n0 >=1

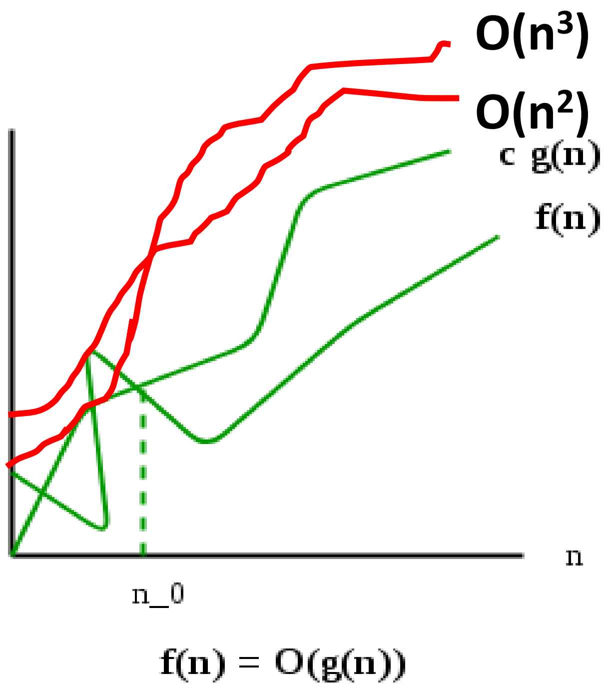


f(n) = O(g(n))

i.e. Cg(n) is an upper bound on the running time of function f(n)

i.e. Function f(n) will never take longer execution time than Cg(n)

Eg.



If $f(n) <= Cg(n)$, $\quad$ => $f(n) = O(n)$ $\quad$ Read as F of n is big Oh of n.

Then $\quad f(n) <= Cg(n^2)$ => $f(n) = O(n^2)$

$f(n) <= Cg(n^3)$ => $f(n) = O(n^3)$

$f(n) <= Cg(n^4)$ => $f(n) = O(n^4)$

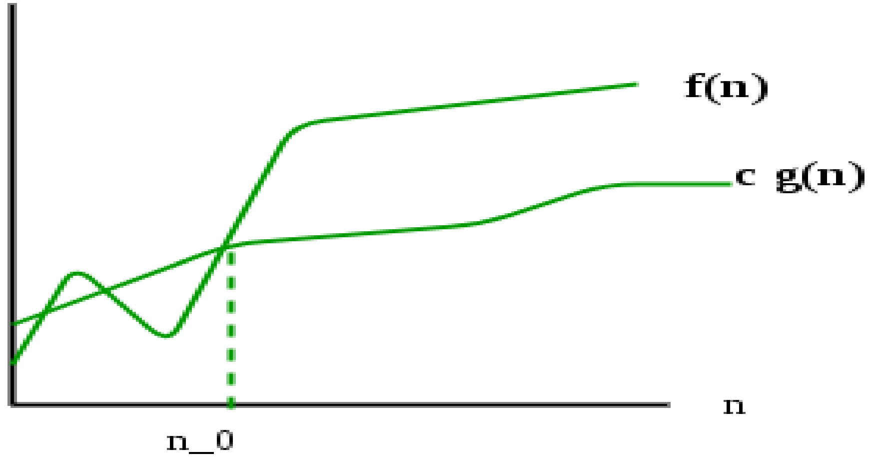
⇒For Big O, more than one bounds are possible , but we always choose the tightest bound.

Thus, $\quad f(n) = O(n)$

2. Big Omega (Ω = Lower Bound = Best case time) :

gives the minimum time required for execution of the algorithm.

i.e. it is the asymptotic lower bound on the running time of an algorithm for any input.



f(n) = Omega(g(n))

$f(n)$ is said to be $\Omega(g(n))$

if there exist a positive real constant C and a positive integer n_0 such that

$f(n) >= Cg(n) > = 0$ For all $n >= n_0$

$\Rightarrow$ We can bound the function f(n) by another function Cg(n) , such that after some value n0 , the value of f(n) is always greater than or equal to Cg(n).

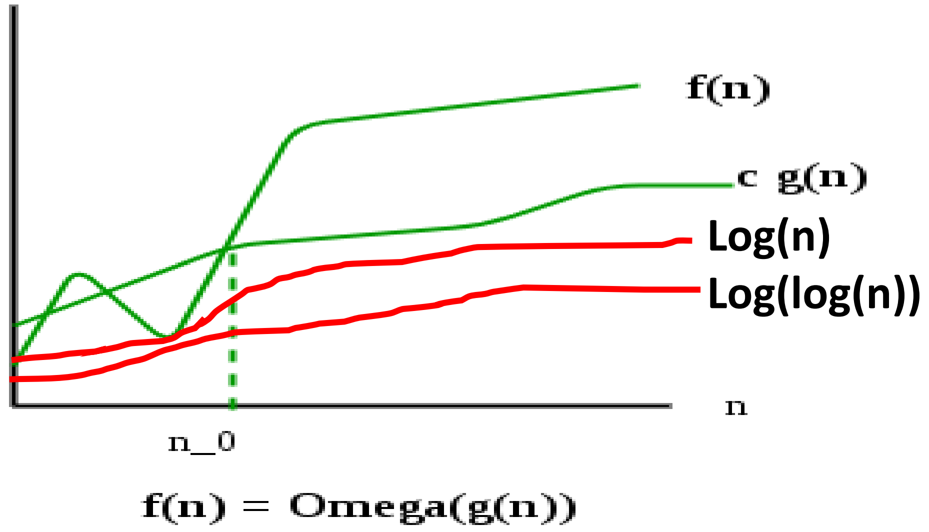
For n >= n0

C > 0

n0 >=1

and hence Cg(n) is the lower bound on the running time of function f(n)

Eg.



f(n) = Ω (g(n)) = Ω(n)

If f(n) is lower bounded by n i.e. Ω (n)

then f(n)) is lower bounded by log n (= less than n) i.e. Ω(log n)

f(n)) is lower bounded by log(log n) (= less than n) i.e. Ω (log(log n))

|

=> For both Big O and Ω , more than one bounds are possible , but we always choose the tightest bound. Thus, f(n) = Ω (g(n))

Big Theta (Θ = Average case time) :

Θ is also know as asymptotically equal bound or asymptotically tight bound.

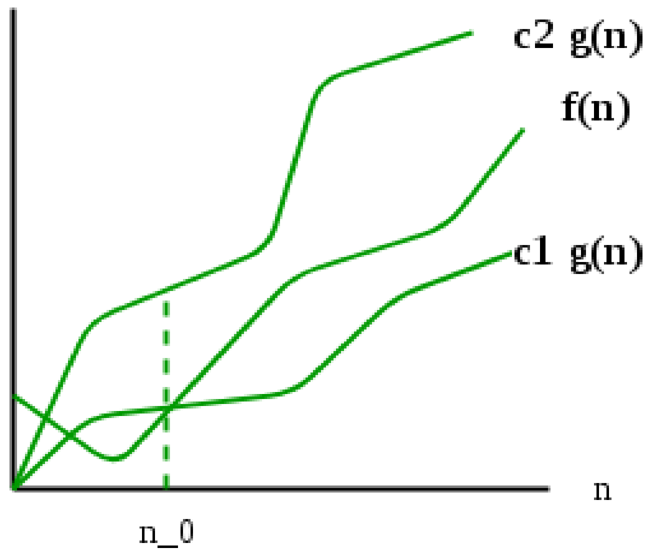
$f(n)$ is $\Theta(g(n))$

iff there exist positive real constants C_1 and C_2 and

a positive integer n_0, such that

$C_1 g(n) <= f(n) <= C_2 g(n)$ for all $n >= n_0$,

where n0>= 1 and C1, C2 >0.



f(n) = theta(g(n))

For all values of n >= n0, the value of f(n) lies at or above C1.g(n) and at or below C2.g(n).

In other words, for all n>= n0, the function f(n) is equal to g(n) to within a constant factor.

We say that g(n) is an asymptotically tight bound for f(n) i.e $f(n) = \Theta(g(n))$.

Actually, for average case,
We just check the leading term of the equation of f(n).

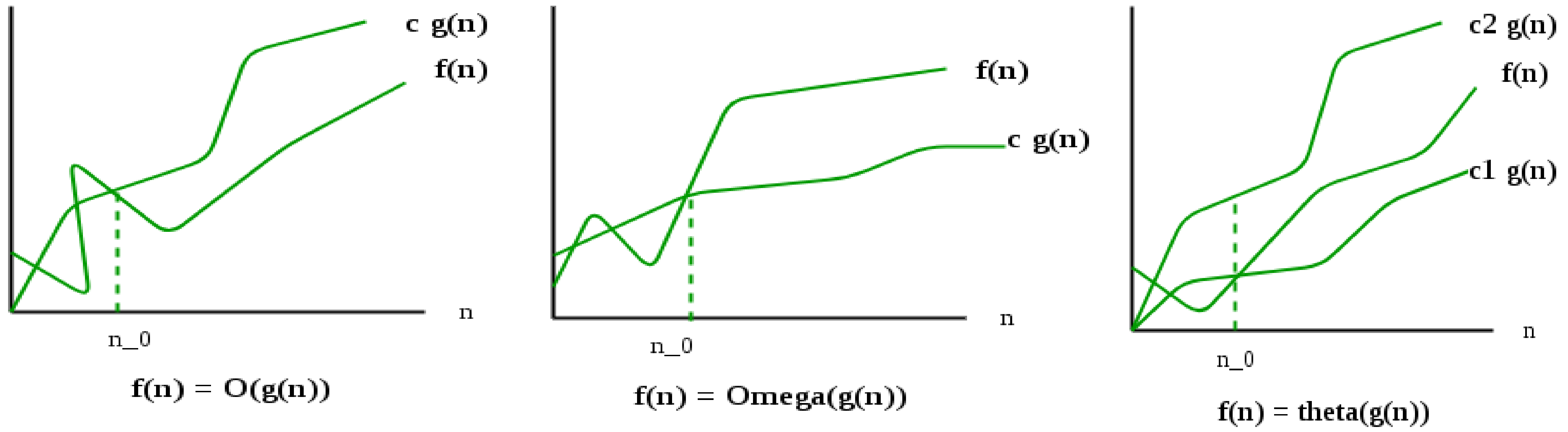
If leading term of $f(n) = n$ then $g(n) = n$ — **Eg. $F(n) = 3n + 1 \Rightarrow \Theta(n)$**

If leading term of $f(n) = n^2$ then $g(n) = n^2$ — **Eg. $F(n) = 3n^2 + n + 1 \Rightarrow \Theta(n^2)$**

If leading term of $f(n) = n^3$ then $g(n) = n^3$ — **Eg. $F(n) = 4n^2 + 2n^3 + n + 1 \Rightarrow \Theta(n^3)$**

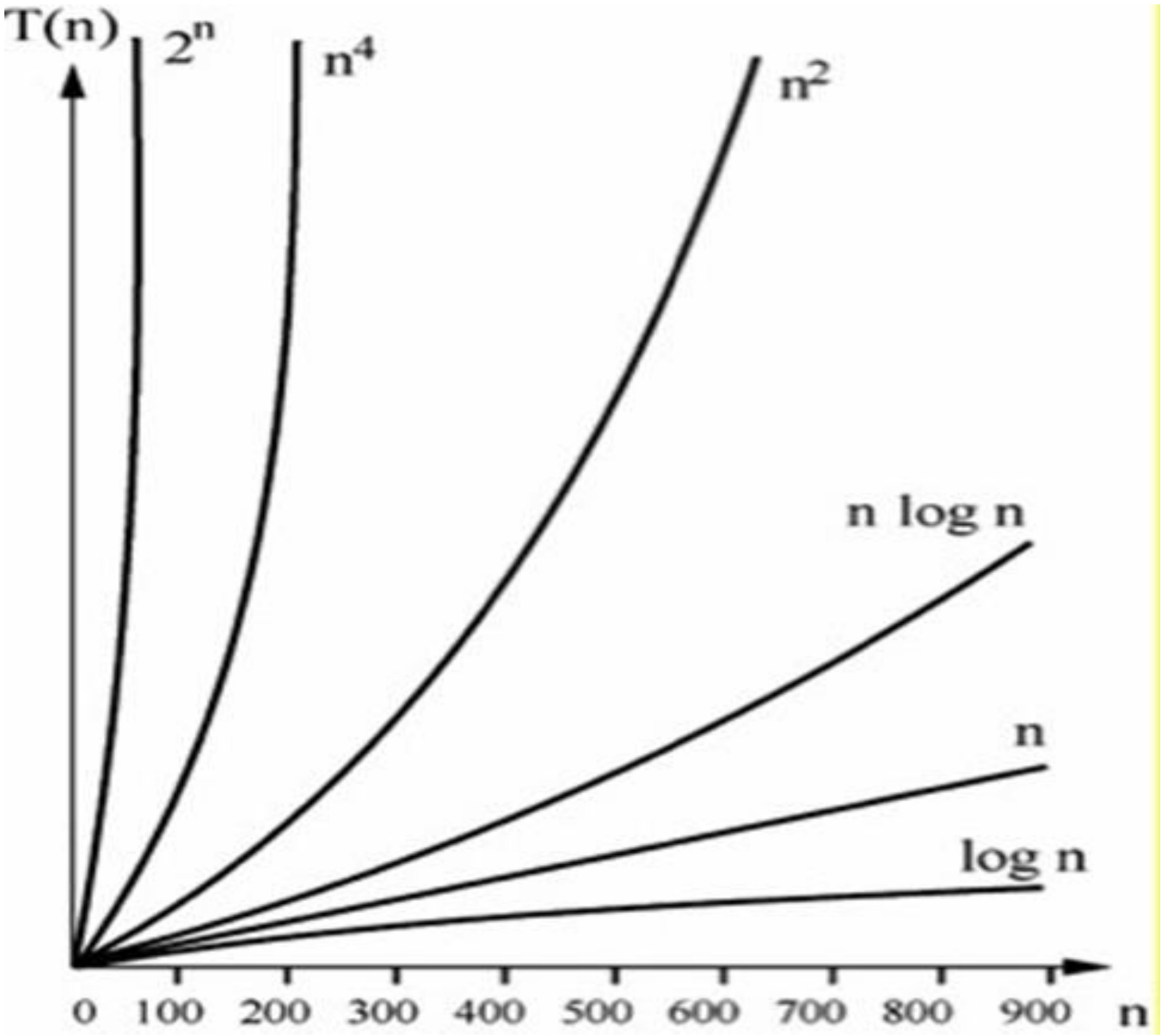
Asymptotic notations used commonly in performance analysis to represent the complexity of algorithms are :

For $n >= n0$, $C > 0$, $n0 >=1$



1. Big Oh (O = Upper Bound = Worst case time) : $0 <= f(n) <= Cg(n)$ For all $n >= n_0$
2. Big Omega (Ω = Lower Bound = Best case time) : $f(n) >= Cg(n) > = 0$ For all $n >= n_0$
3. Big Theta (Θ = Average case time) : $C_1 g(n) <= f(n) <= C_2 g(n)$ For all $n >= n_0$

is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n\log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	Polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential
2^10 = 1024	2^17 = 1, 31,072



Module 1 : Introduction

Last Session :

Asymptotic notations :

1. Big Oh (O = Upper Bound = Worst case time)
2. Big Omega (Ω = Lower Bound = Best case time)
3. Big Theta (Θ = Average case time)

Current Session :

Different types of Algorithms

Iterative algorithms

Recursive algorithms

Time complexity of Iterative algorithms

Time complexity of algorithms

Iterative algorithms

```
F(n)
{
  for I = 1 to n
      print (" Hello")
}
```

Time Complexity :

Count No. Of times, the loop gets executed.

T(n) = O(n)

Recursive algorithms

```
F(n)
{
  if( condition)   n>1
      F(n/2) or F (n-1)
}
```

Time Complexity :

Represent F(n) in terms of F(n/2) or F (n-1) ..

No Iteration & No Recursion

C = a + b
b = a / 2
d = sqrt(a)

Time required is always a constant time = O(1) for any input size.

Every recursive solution can be implemented as an iterative solution and vice versa.

Both approaches are equal in power.

Time complexity of Iterative algorithms :

1. For (I = 1 to n , I++)
   print (" Hello") => "Hello" gets printed n times.

Big Oh = Worst case time
= Upper Bound
= $O(n)$

2. For (I = 1 to n , I++)
   For (J = 1 to n , J++)
   print (" Hello")

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

3. For (I = 1 to n , I++)
   For (J = 1 to n , J++)
   For (K = 1 to n , K++)
   print (" Hello")

= $O(n * n * n)$
= $O(n^3)$

Time complexity of Iterative algorithms :

For (i = 1; i < n ; i =i * 2) = $O(\log_2 n)$

print (" Hello")

i = 1 2 4 8 16 32 64 .... n

2^0 2^1 2^2 2^3 2^4 2^5 2^6 ....< n

n = 10 Loop Iterates 4 times

n = 20 Loop Iterates 5 times

n = 30 Loop Iterates 5 times

n = 40 Loop Iterates 6 times

Let K be the number of times the loop gets executed before we reach the value of n,

Then in above example,

$2^K <= n$

$2^3 <= 10$

$2^4 <= 20$

$2^4 <= 30$

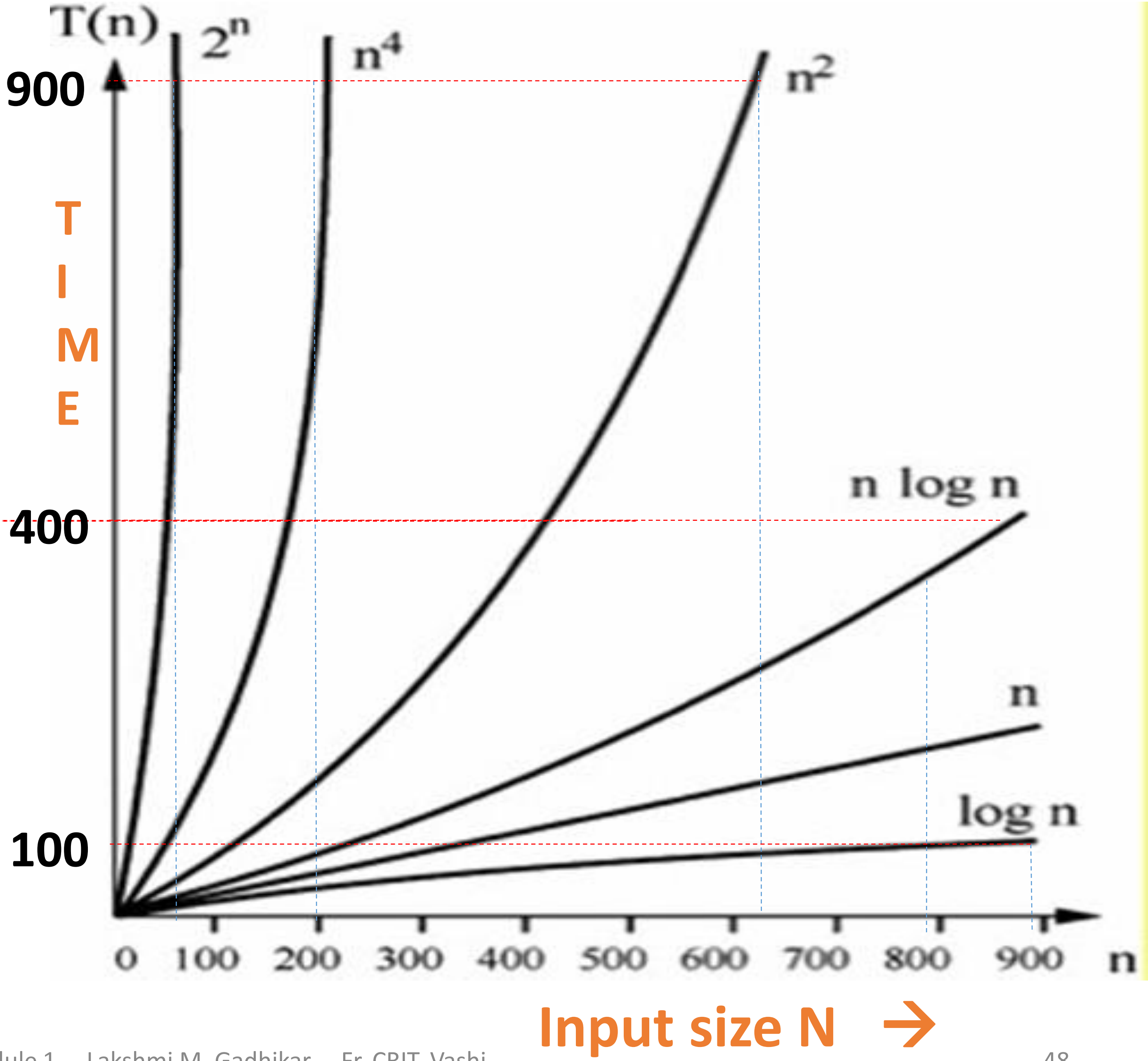
Thus, in this example, the loop is not executed for K iterations,

It is executed for (K+1) iterations.

$2^{K+1} = n \Rightarrow (K+1) = \log_2 n$

$= O(\log_2 n)$

is proportional to:	Complexity:
$T(n) \propto \log n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n\log n$	linearithmic
$T(n) \propto n^2$	quadratic
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	Polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n; k > 1$	exponential



Time complexity of Iterative algorithms :

For (i = 1; i < n ; i =i * 2) $= O(\log_2 n)$

5. For (i = n/2 ; i <= n ; i ++)
   For (j = 1 ; j <= n ; j = 2 * j)
   For (k = 1 ; k <= n ; k = k * 2)
   print (" Hello")

i Loop Iterates n/2 times

j Loop Iterates $\log_2 n$ times

k Loop Iterates $\log_2 n$ times

Thus, total number of iterations $= n/2 \ . \log_2 n \ . \log_2 n$

$= n/2 \, (\log_2 n)^2$

$= O(n \, (\log_2 n)^2)$

6. For (i = n/2 ; i <= n ; i ++)

   For (j = 1 ; j <= n ; j = 2 * j)

   For (k = 1 ; k <= n ; k = k * 2)
   print (" Hello")

i Loop Iterates n/2 times

j Loop Iterates $\log_2 n$ times

k Loop Iterates $\log_2 n$ times

Thus, total number of iterations =

$n/2 + \log_2 n + \log_2 n$

$= n/2 + 2(\log_2 n)$ $= O(n)$

Quiz :

Q1. State whether the statement "**Every recursive solution can be implemented as an iterative solution and vice versa**" **is TRUE/FALSE**

Q2. Find the Time complexity of following Iterative algorithms :

```
For ( i = n/2 ;  i <= n ; i ++ )          ------------------------n/2
    For ( j = 1 ;  j <= n/2 ; j ++ )       -------------------n/2
        For ( k = 1 ;  k <= n ; k = k * 2 )     ------------Log2n
            print ( " Hello")
```

$$T(n) = n/2 * n/2 * \log_2 n$$
$$= n^2/4 * \log_2 n$$
$$= O(n^2 \log_2 n)$$

options:

$O(n (\log_2 n))$

$O(n^2 (\log_2 n)^2)$

$O(n (\log_2 n)^2)$

$O(n^2 (\log_2 n))$

Advanced Data Structures and Analysis
ITDO 5014

Module 1 : Introduction

Last Session :

Different types of Algorithms

Iterative algorithms

Recursive algorithms

Time complexity of Iterative algorithms

Current Session :

Time complexity of Recursive algorithms

Time complexity of algorithms

Iterative algorithms

```
F(n)
{
  for I = 1 to n
      print (" Hello")
}
```

Time Complexity :

Count No. Of times, the loop gets executed.

$T(n) = O(n)$

Recursive algorithms

```
F(n)
{
  if( condition)
      F(n/2) or F (n-1)
}
```

Time Complexity :

Represent F(n) in terms of F(n/2) or F (n-1) ..

No Iteration & No Recursion

Time required is always a constant time = O(1) for any input size.

Every recursive solution can be implemented as an iterative solution and vice versa.

Both approaches are equal in power.

Time complexity of recursive algorithms

The substitution Method

Recursive tree Method

Masters method

Q. Explain different methods for solving recurrences (10M)

Q. Explain Substitution method for solving recurrences with suitable example (10M)

Q. Explain Recursive tree method for solving recurrences (10M)

Q. Explain Masters methods for solving recurrences (10M)

1) Substitution Method:

A(n) T(n) = Time taken by A(n)

```
A(n)
{
    if ( n > 1 )
        return ( A(n/2) + A(n/2) )
}
```

Lets determine T(n) as a sum of following times :

1. Time required to check if the condition if (n>1) is satisfied = Constant time say C or O(1) .
2. Determine time required for A(n/2)
   A(n/2) is called 2 times
   So, Total time required = 2 . T(n/2)

➔ $T(n) = C + 2\,T(n/2)$

C ➔ Time required for execution of if condition and addition (A(n/2) + A(n/2))

2 T(n/2) = Time required for execution of 2 recursive calls.

This is how we form a recurrence relation which can be solved using any of the above 3 approaches to find the time complexity.

```
A(n)
{
    if ( n > 1 )
        return  A(n -1)
}
```

T(n) = Time taken by A(n)

A(n) calls itself recursively till the terminating condition (n>1) is satisfied.

So, n = 1, which stops recursion, is called as
anchor condition or
base condition or
stop condition.

=>

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

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

Substitute (n-1) in place of n in equation 1

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

Substitute (n-2) in place of n in equation 1

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

Substituting 2 in 1 =>

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

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

Substitute 3 in 4 =>

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

$= 3 + T(n-3)$

|

|

$= k + T(n-k)$

$T(n) = 1 + 1 + T(n-2)$
$T(n) = 1 + 1 + 1 + T(n-3)$
$T(n) = 1 + 1 + 1 + 1 + T(n-4)$
|
$T(n) = k + T(n-k)$ --------- 5

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

Terminating condition is $T(n) = 1$
For terminating, we need to get $T(n-k) = 1$ in equation 5.
$\Rightarrow n - k = 1$
$\Rightarrow k = n - 1$
$\Rightarrow$ For value of $k = n-1$, $T(n-k)$ will become 1.

Substituting this in equation 5, we get,

Eq. 5 => $T(n) = k + T(n-k)$
$= (n-1) + T(n-(n-1))$
$= (n-1) + T(1)$

$$T(n) = \begin{cases} 1 + T(n-1) & \text{for } n > 1 \\ 1 & \text{for } n = 1 \end{cases}$$

Q. Solve the following recurrence relation using substitution method.

$$T(n) = n + T(n-1) \quad \text{For } n > 1$$
$$= 1 \text{ or } C \quad \text{For } n = 1$$

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

Substitute (n – 1) in place of n in equation 1

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

$\Rightarrow T(n-2) = (n-2) + T(n-3)$ --------3

Example: $T(n) = n + T(n-1)$ for $n > 1$

$= 1$ or C for $n = 1$

Substituting 2 in 1 =>

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

Substitute 3 in 4 =>

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

|

|

$= n + (n-1) + (n-2) + \text{----------} + (n-k) + T(n-(k+1)) = 1$ $\boxed{\text{i.e. } T(1) = 1}$ -- 5

i.e. $n - (k+1) = 1$ => $n-k-1 = 1$ => $\boxed{k = n-2}$

We get T(1) for value of $k = n - 2$ and we stop recursion

Substitute $k = n - 2$ in equation 5

$T(n) = n + T(n-1)$ for $n > 1$
$= 1$ for $n = 1$
$\Rightarrow T(1) = 1$

We get T(1) for value of $k = n - 2$ and we stop recursion
Substitute $k = n - 2$ in equation 5

Eq 5 $\Rightarrow T(n) = n + (n-1) + (n-2) + \text{----------} + (n - k) + T(n - (k+1)) = 1$ i.e. $T(1) = 1$

$$T(n) = n + (n-1) + (n-2) + \text{----------} + (\cancel{n} - (\cancel{n} - 2)) + T(\cancel{n} - ((\cancel{n} - 2) + 1))$$

$T(n) = n + (n-1) + (n-2) + \text{----------} + 2 + 1 \Rightarrow$ Sum of first n natural numbers $= n(n+1)/2$

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

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

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

Example : $T(n) = n + T(n-1)$ **For** $n > 1$

$= 1$ **For** $n = 1$

Summary :

Time complexity of recursive algorithms

- **The substitution Method**
- **Recursive tree Method**
- **Masters method**

Recurrences:

The substitution Method

Recursive tree Method

Masters method

Recurrences - Find time complexity using the Recursive tree Method :

Example : $T(n) = 2\,T(n/2) + C$; $n > 1$

$T(1) = C$ $n = 1$

$= O(n)$

T(n) ---------- C ------------------------------ C

T(n/2) C C T(n/2) ---- 2C

T(n/4) C C T(n/4) T(n/4) C C T(n/4)-- 4C

T(n/8) C C T(n/8) C C C C C C-- 8C

| |

| |

T(n/n) T(n/n) -- nC

T(1) T(1) ---

Total work done = C + 2C + 4C + 8C + --------- + nC

l=0 h=5

8	2	17	5	9	1

```
Void A1 (int arr[n], int l, int h)        T(n)
{
    if ( m > 1 )
    {                                      C
        int m = ( l + h ) / 2;

        // Divide array in two halves
        A1 ( arr,  l,  m ) ;               T(n/2)
        A1 ( arr,  m+1,  h ) ;             T(n/2)
    }
}
```

Total work done = 1C + 2C + 4C + 8C + --------- + nC

Lets assume that n is some power of 2 i.e. $n = 2^k$

Total work done = 1C + 2C + 4C + 8C + --------- + nC

Total work done = $C2^0 + C2^1 + C2^2 + C2^3 + \text{---------} + C2^k$

Geometric series : $a + ar + ar^2 + ar^3 + \text{---------} + ar^{(n-1)}$

$$\boxed{= \sum_{k=0}^{n-1} (a.\, r^k) = a \left(\frac{1 - r^n}{1 - r} \right)} \Rightarrow = C. \left(\frac{(1-2^{k+1})}{(1-2)} \right)$$

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

$= C\ (2^k * 2 - 1)$ $\quad 2^k = n$

$= C\ (n*2 - 1)$

$= 2C*\ n - C$

$= O(n)$

a = First Term = C

r = Common ratio between terms = 2

n = Number of terms = k+1

Summary : To find time complexity using the Recursive tree Method

$$T(n) = 2\,T(n/2) + C \; ; \quad n > 1$$
$$= C \quad n = 1$$
$$= O(n)$$

T(n) ---------- C -------------------------- C

T(n/2) C C T(n/2) ---- 2C

T(n/4) C C T(n/4) T(n/4) C C T(n/4)-- 4C

T(n/8) C C T(n/8) C C C C C C-- 8C

|

|

T(n/n) T(n/n) ------------------------------------ nC

T(1) T(1) ------------------------------------

Total work done = C + 2C + 4C + 8C + --------- + nC

Recurrences:

- The substitution Method
- Recursive tree Method
- Masters method

Masters Theorem : Master Method is a direct way to get the solution.

The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$ where $a >= 1$, $b > 1$, $k>= 0$, p is a real number.

There are following three cases:

1. If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$

2. If $a = b^k$ then
   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 $a < b^k$ then
   a. If $p >= 0$ then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$ then $T(n) = O(n^k)$

Masters Theorem :

Q. Solve the following recurrence relation using master's method (5M)

1. $T(n) = 9\,T(n/3) + n$

Solution :

$T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$ where $a >= 1$, $b > 1$, $k>= 0$, p is a real number.

$a = 9 \quad b = 3 \quad k = 1 \quad p = 0$

$a \;\; ? \;\; b^k$

$9 \;\; ? \;\; 3^1; \quad 9 > 3 \,; \quad a > b^k$ ➔ Condition 1 of Master's Theorem.

1. If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$

➔ $T(n) = \Theta(n^{\log_3 9})$: $\text{Log}_3 9 = 2$; $3^? = 9$; $? = 2$

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

****

Masters Theorem :

Solve the following recurrence relation using master's method

2. $T(n) = 16\,T(n/4) + n^2$

Solution :

$T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$ where $a >= 1$, $b > 1$, $k >= 0$, p is a real number.

$a = 16$ $\quad b = 4$ $\quad k = 2$ $\quad p = 0$

a ? b^k

16 ? 4^2; $16 = 16$; $a = b^k$ and $p = 0 > -1$ ➔ Condition 2a of Master's Th.

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

2. If $a = b^k$ then
   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 $T(n) = \Theta(n^{\log_b a})$

➔ $T(n) = \Theta(n^{\log_4 16} \log^1 n)$ $\quad$: $\log_4 16 = 2$; $4^? = 16$; $? = 2$

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

****

Masters Theorem :

Solve the following recurrence relation using master's method

3. $T(n) = 3\ T(n/2) + n^2$

Solution :

$T(n) = a\ T(n/b) + \Theta\ (n^k \log^p n)$ where $a >= 1$, $b > 1$, $k >= 0$, p is a real number.

$a = 3 \quad b = 2 \quad k = 2 \quad p = 0$

$a \quad ? \quad b^k$

$3 \quad ? \quad 2^2; \quad 3 < 4; \quad a < b^k$ and $p = 0$ ➔ Condition 3a of Master's Th.

3. If $a < b^k$ then
   a. If $p >= 0$ then $T(n) = \Theta\ (n^k \log^p n)$

3. If $a < b^k$ then
   a. If $p >= 0$ then $T(n) = \Theta\ (n^k \log^p n)$
   b. If $p < 0$ then $T(n) = O\ (n^k)$

➔ $T(n) = \Theta\ (n^2 \log^0 n)$

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

****

Masters Theorem :

Solve the following recurrence relation using master's method

4. $T(n) = 8\,T(n/2) - n^2$

Solution :

Master's Theorem can not be applied for above problem.

Master's Theorem :

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$ where $a >= 1$, $b > 1$, $k >= 0$, p is a real number.

There are following three cases:

1. If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$

2. If $a = b^k$ then
   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 $a < b^k$ then
   a. . If $p >= 0$ then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$ then $T(n) = O(n^k)$

Masters Theorem :

Solve the following recurrence relations using master's method

5. $T(n) = 2\,T(n/2) + n \log n$

Solution :

$T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$ where $a >= 1$, $b > 1$, $k >= 0$, p is a real number.

$a = 2$ $b = 2$ $k = 1$ $p = 1$

a ? 2^k

2 ? 2^1; $2 = 2$; $a = b^k$ and $p = 1 > -1$ ➔ Condition 2a of Master's Th.

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

➔ $T(n) = \Theta(n^{\log_2 2} \log^2 n)$: $\log_2 2 = 1$; $2^? = 2$; $? = 1$

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

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

Masters Theorem :

Solve the following recurrence relations using master's method

5. $T(n) = T(2n/3) + 1$ (Cormen Pg 95)

Solution :

$T(n) = a\,T(n/b) + \Theta(n^k \log^p n)$ where $a >= 1$, $b > 1$, $k >= 0$, p is a real number.

$a = 1 \quad b = 3/2 \quad k = 0 \quad p = 0$

a ? 2^k

1 ? $(3/2)^0$; $1 = 1$; $a = b^k$ and $p = 0 > -1$ ➔ Condition 2a of Master's Th.

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

➔ $T(n) = \Theta(n^{\log_{3/2} 1} \log^1 n)$: $\log_{3/2} 1 = 0$; $(3/2)^? = 1$; $? = 0$

$T(n) = \Theta(n^0 \log n)$

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

Masters Theorem : **Master Method is a direct way to get the solution.**

The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$ where $a >= 1$, $b > 1$, $k >= 0$, p is a real number.

There are following three cases:

1. If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$

2. If $a = b^k$ then
   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 $a < b^k$ then
   a. . If $p >= 0$ then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$ then $T(n) = \Theta(n^k)$

Masters Theorem :

Assignment 1 : (20 M)

Solve the following recurrence relations using master's method

1. $T(n) = 3\,T(n/2) + n^2$
2. $T(n) = 16\,T(n/4) + n$
3. $T(n) = 2\,T(n/2) + n \log n$
4. $T(n) = 2\,T(n/2) + n / \log n$

Quiz6

Q1. Masters method is used to find the time complexity of recurrence relations of the form
$T(n) = aT(n/b) + \Theta(n^k \log^p n)$ where $a < 1$, $b > 1$, $k >= 0$, p is a real number.
TRUE/FALSE

Q2. Find the time complexity of

$$T(n) = 4\,T(n/2) + \log n$$

Options :

$T(n) = O(n)$

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

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

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

Space Complexity:

Space Complexity : of an algorithm is total main memory space taken by the algorithm with respect to the input size.
i.e. Space complexity includes both Auxiliary space and space used by input.

The term Space Complexity is misused for Auxiliary Space at many places.

Auxiliary Space : is the extra space or temporary space used by an algorithm.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity.
(as all sorting algorithms require same input array of size n)
Sort (int a[n], int n)

Merge Sort uses O(n) auxiliary space,

```
Sort ( int a[n], int n)
  { int a[n]
    ---- }
```

Insertion sort and Heap Sort use O(1) auxiliary space.

Space complexity of all these sorting algorithms is O(n).

In such cases, we use auxiliary space for comparison.

Thus, space complexity is a measure of the amount of working storage an algorithm needs.

That means how much memory, in the worst case, is needed at any point in the algorithm.

As with time complexity, space complexity of an algorithm indicates how the space needs grow, in big-Oh terms, as the size N of the input problem grows.

1.

```
int sum(int x, int y, int z)
{
  int r = x + y + z;
  return r;
}
```

Space complexity = Parameter Space + Auxiliary Space

Parameter Space = 3 Units for 3 parameters x, y, z.

Auxiliary space = 1 (for variable r)

Total Space = Algo requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so this is O(1).

Space complexity = 3+1 = 4 = constant value = O(1)

2.

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

Space complexity = Parameter Space + Auxiliary Space

Parameter Space = n units of space for array a and 1 unit of space for the variable n = n+1.

Auxiliary space = 2 (1 for variable r and 1 for var i)

Total Space i.e.

Space complexity = n + 1 + 2 = n+3 = O(n)

Q. Determine the space complexity of following algorithms.

1. A(int a[n], int n)
```
{
        int I;
        for ( I = 1 to n )
            a[i] = 0;
}
```

2. A(int a[n], int n)
```
{
        int I =0;  j  =20
        for ( I = 1 to j )
            a[i] = 0;
}
```

3. A(int a[n], int n)
```
{
        int I;
        float B[n]
        for ( I = 1 to n )
            B[i] =  a[i] ;
}
```

4. A(int a[n], int n)
```
{
        int I, j;
        int B[n][n]
        for ( I = 1 to n )
            for ( j = 1 to n )
                B[i][j] =  a[i] ;
}
```

Q. Determine the space complexity of following algorithms.

1. A(int a[n], int n)

```
{
        int I;
        for ( I = 1 to n )
            a[i] = 0;
}
```

Parameter Space = n + 1

Auxiliary Space = 1

Total Space = Space Complexity = Parameter Space + Auxiliary Space

= n+1+1

= n +2 = O(n)

---

2. A(int a[n], int n)

```
{
        int I =0;  j  =20
        for ( I = 1 to j )
            a[i] = 0;
}
```

Parameter Space = n + 1

Auxiliary Space = 1 + 1 = 2

Total Space = Space Complexity = Parameter Space + Auxiliary Space

= n+1+2

= n +3 = O(n)

Q. Determine the space complexity of following algorithms.

3. A(int a[n], int n)

```
{
        int I;
        float B[n]
        for ( I = 1 to n )
            B[i] =  a[i] ;
}
```

Parameter Space = n + 1

Auxiliary Space = 1 + n

Total Space = Space Complexity = Parameter Space + Auxiliary Space

= n+1 + 1+n

= 2n +2 = O(n)

4. A(int a[n], int n)

```
{
        int I, j;
        int B[n][n]
        for ( I = 1 to n )
            for ( j = 1 to n )
                B[i][j] =  a[i] ;
}
```

Parameter Space = n + 1

Auxiliary Space = 2 + (n * n) = 2 + n^2

Total Space =

Space Complexity = Parameter Space + Auxiliary Space

= n+1 + 2+n^2

= n^2 + n + 3 = O(n^2)

```
A()
{
    int a = 0, b = 0;
    for (int i = 0; i < N; i++)
    {
        a = a + rand();
    }
    for (int j = 0; j < M; j++)
    {
        b = b + rand();
    }
}
```

Time Complexity = O(N + M) time

Space Complexity = O(1) space

Time Complexity : The first loop is O(N) and the second loop is O(M).

Since we don't know which is bigger, we take write it as O(max (N, M)).

Time Complexity = O(N + M) time

Space Complexity :

Total Space = Space Complexity = Parameter Space + Auxiliary Space

Parameter Space = 0

Auxiliary Space = 1 for var a + 1 for var b + 1 each for I and j =
= 1+1+1+1 = 4

Total Space = Space Complexity = Parameter Space + Auxiliary Space
= 0+4 = 4 = Constant space = O(1)

Since there is no additional space being utilized, the space complexity is constant = O(1)

```
A()
{
    int a = 0, b = 0;
    for (int i = 0; i < N; i++)
    {
        a = a + rand();
    }
    for (int j = 0; j < M; j++)
    {
        b = b + rand();
    }
}
```

Time Complexity = O(N + M) time

Space Complexity = O(1) space

Time Complexity : The first loop is O(N) and the second loop is O(M).

Since we don't know which is bigger,
we say this is O(N + M).

This can also be written as O(max(N, M)).
Time Complexity = O(N + M) time

Space Complexity :
Total Space = Space Complexity = Parameter Space + Auxiliary Space

Parameter Space = 0
Auxiliary Space = 1 for var a + 1 for var b + 1 each for I and j =
= 1+1+1+1 = 4

Total Space = Space Complexity = Parameter Space + Auxiliary Space
= 0+4 = 4 = Constant space = O(1)

Since there is no additional space being utilized, the space complexity is constant = O(1)

Thank you