SPPU New Syllabus

A Book Of

Design and Analysis of Algorithms

For M.Sc. (Computer Science) : Semester - II

[Course Code : CS-551-MJ : Credits-04]

CBCS Pattern

CBCS as per New Syllabus (Under NEP), Effective from 2023-2024

Mrs. Monica Deshmane (Dhotre)
MCS, UGC NET, Pursuing Ph.D.
Asst. Professor
Haribhai V. Desai College of Commerce, Arts and Science
Pune 411002

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N1171

DESIGN AND ANALYSIS OF ALGORITHMS **ISBN 978-93-6109-764-5**

First Edition : January 2024

© : Author

The text of this publication, or any part thereof, should not be reproduced or transmitted in any form or stored in any computer storage system or device for distribution including photocopy, recording, taping or information retrieval system or reproduced on any disc, tape, perforated media or other information storage device etc., without the written permission of Author with whom the rights are reserved. Breach of this condition is liable for legal action.

Every effort has been made to avoid errors or omissions in this publication. In spite of this, errors may have crept in. Any mistake, error or discrepancy so noted and shall be brought to our notice shall be taken care of in the next edition. It is notified that neither the publisher nor the author or seller shall be responsible for any damage or loss of action to any one, of any kind, in any manner, therefrom. The reader must cross check all the facts and contents with original Government notification or publications.

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041

DISTRIBUTION CENTRES

PUNE

Nirali Prakashan
(For orders outside Pune)
S. No. 28/27, Dhayari Narhe Road, Near Asian College
Pune 411041, Maharashtra
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

Nirali Prakashan
(For orders within Pune)
119, Budhwar Peth, Jogeshwari Mandir Lane
Pune 411002, Maharashtra
Mobile : 9657703145, 9890997937
Email : niralilocal@pragationline.com

MUMBAI

Nirali Prakashan
Rasdhara Co-op. Hsg. Society Ltd., 'D' Wing Ground Floor, 385 S.V.P. Road
Girgaum, Mumbai 400004, Maharashtra
Mobile : 7045821020, Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES

DELHI

Nirali Prakashan
Room No. 2 Ground Floor
4575/15 Omkar Tower, Agarwal Road
Darya Ganj, New Delhi 110002
Mobile : 9555778814/9818561840
Email : delhi@niralibooks.com

BENGALURU

Nirali Prakashan
Maitri Ground Floor, Jaya Apartments,
No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003
Karnataka; Mob : 9686821074
Email : bengaluru@niralibooks.com

NAGPUR

Nirali Prakashan
Above Maratha Mandir, Shop No. 3,
First Floor, Rani Jhanshi Square,
Sitabuldi Nagpur 440012 (MAH)
Tel : (0712) 254 7129
Email : nagpur@niralibooks.com

KOLHAPUR

Nirali Prakashan
New Mahadvar Road, Kedar Plaza,
1st Floor Opp. IDBI Bank
Kolhapur 416 012 Maharashtra
Mob : 9850046155
Email : kolhapur@niralibooks.com

JALGAON

Nirali Prakashan
34, V. V. Golani Market, Navi Peth,
Jalgaon 425001, Maharashtra
Tel : (0257) 222 0395
Mob : 94234 91860
Email : jalgaon@niralibooks.com

SOLAPUR

Nirali Prakashan
R-158/2, Avanti Nagar, Near Golden
Gate, Pune Naka Chowk
Solapur 413001, Maharashtra
Mobile 9890918687
Email : solapur@niralibooks.com

marketing@pragationline.com | www.niralibooks.com

Also find us on www.facebook.com/niralibooks

Preface ...

"Design and Analysis of Algorithms" is a fundamental subject of Computer Science which is designed to become expertise in designing algorithms in most efficient ways by considering its time and space complexity. Due to this student can easily write code in any programming language. It is my pleasure to present book on this subject for **M.Sc. (Computer Science): Semester -2 as per NEP syllabus at 2023 pattern**.

To make subject very interesting, there are lot of comparative techniques used such as Divide and Conquer, Greedy, Dynamic programming, Backtracking, Branch and Bound, Problem classifications. Each chapter contains simple language for explanation and examples are given with tracings of an algorithms as well as programs. At the end of each Chapter, MCQs, Short answer questions and Long answer questions are provided for practice.

I, Monica Dhotre, am so thankful from the bottom of my heart to **Parampujya Shree Sadguru Narayan Maharaj (Anna Maharaj) of Shree Kshetra Narayanpur**, Tal-Purandar, District-Pune for blessing me with more than I deserve.

I am thankful to my mother Mrs. Meera Deshmane, My husband Sagar Dhotre , my son Smaran, my mother-in-law Mahananda, my brother Dheeraj, all my sisters and my whole family members for their support and inspiration.

I also thankful to my friends, collegues for their support.

I express my sincere thanks to our former principal Dr. Ganesh Raut sir, Dr. Gulab Gugale sir (Chemistry Dept. H.V. Desai College, Pune) for their support in writing book.

My sincere thanks to Shri. Dineshbhai Furia, Mr. Jignesh Furia for trusting and giving this opportunity to work with them.

We also thank to Mrs. Anita Panajkar, Ms. Madhuri Gabale, Mr. Akbar Shaikh, Ms. Chaitali Takale for their excellent co-operation.

We also thanks to Nirali Prakashan for bringing out the book in the market within short period of time.

Any suggestions and improvements in contents of book will be appreciated and accordingly modifications can be done in later editions.

Asst. Prof. Monica Deshmane (Dhotre)
Haribhai V. Desai College of Arts, Science and Commerce, Pune

■■■

Syllabus ...

1. **Basics of Algorithms** **[6 Hrs.]**
   - Algorithm Definition and characteristics
   - Space Complexity
   - Time Complexity, Worst Case, Best Case, Average Case
   - Complexity, Asymptotic Notation
   - Recursive and Non-recursive Algorithms
   - Sorting Algorithms: insertion sort, heap sort, bubble sort
   - Sorting in Linear Time: Counting Sort, Concept of Bucket and Radix Sort
   - Searching Algorithms: Linear, Binary
2. **Divide and Conquer Strategy** **[10 Hrs.]**
   - General Method, Control Abstraction
   - Binary Search
   - Merge Sort, Quick Sort
   - Comparison between Traditional Method of Matrix Multiplication vs. Strassen's Matrix Multiplication
   - Writing simple algorithm using Divide and conquer strategy: power(x,n), find occurrence of a number from array of N integers, to find minimum from an array, mini-max algorithm, largest number multiplication, simple convex algorithm
3. **Greedy Method** **[10 Hrs.]**
   - Knapsack Problem
   - Job Sequencing with deadlines
   - Minimum-cost Spanning Trees: Kruskal and Prim's Algorithm
   - Optimal Storage on Tapes
   - Optimal Merge Patterns
   - Huffman Coding
   - Shortest Path: Dijkstra's Algorithm
4. **Dynamic Programming** **[12 Hrs.]**
   - Principle of Optimality
   - Matrix Chain Multiplication
   - 0/1 Knapsack Problem:
     (i) Merge and Purge
     (ii) Functional Method
   - Bellman Ford Algorithm
   - All Pairs Shortest Path Floyd - Warshall Algorithm
   - Travelling Salesperson Problem
   - Longest Common Subsequence
   - String editing

5. Decrease and Conquer **[6 Hrs.]**

- Definition of Graph Representation of Graph
- By Constant - DFS and BFS
- Topological Sorting
- Strongly Connected Components and Spanning Trees
- Articulation Point and Bridge Edge

6. Backtracking **[7 Hrs.]**

- General Method
- Fixed Tuple vs. Variable Tuple Formulation
- n-Queen's Problem
- Graph Colouring Problem
- Hamiltonian Cycle
- Sum of Subsets

7. Branch and Bound Technique **[6 Hrs.]**

- Introduction: Branch and bound terms like definition of live node, E-node, Dead node, Least cost (LC) search, Least cost Branch and Bound (LCBB)
- 0/1 Knapsack Problem Using LCBB method (fixed tuple size)
- Traveling Salesman Problem using LCBB Method (Variable tuple size)

8. Problem Classification **[3 Hrs.]**

- The Class of P, NP, NP-hard and NP - Complete
- Relationship among P class, NP class, NP-hard and NP -Complete
- Cook's Theorem

■■■

Contents ...

1.	Basics of Algorithms	1.1 – 1.41
2.	Divide and Conquer Strategy	2.1 – 2.21
3.	Greedy Method	3.1 – 3.34
4.	Dynamic Programming	4.1 – 4.34
5.	Decrease and Conquer	5.1 – 5.37
6.	Backtracking	6.1 – 6.26
7.	Branch and Bound	7.1 – 7.28
8.	Problem Classification	8.1 – 8.16

■■■

CHAPTER 1

Basics of Algorithms

Contents

1.1 Introduction
- 1.1.1 Algorithm Definition and Characteristics
- 1.1.2 Analyzing Algorithm
- 1.1.3 Algorithm Specification (Pseudo Code Convention)
- 1.1.4 Algorithm Types
- 1.1.5 Examples of Algorithm

1.2 Complexity Analysis: Space Complexity

1.3 Complexity Analysis: Time Complexity

1.4 Step Count: Analysis of Complexity

1.5 Asymptotic Notation
- 1.5.1 Types of Asymptotic Notation
  - 1.5.1.1 Big O (Big Oh Notation) (called Upper Bound)
  - 1.5.1.2 Big Omega Notation Ω (called Lower Bound)
  - 1.5.1.3 Big Theta Notation: Θ (Asymptotic Tight Bound)
- 1.5.2 Rank Ordering

1.6 Recursive Algorithms
- 1.6.1 Tower of Hanoi
- 1.6.2 Recursion Types

1.7 Permutation Generator

1.8 Sorting Algorithms
- 1.8.1 Sorting in Non-Linear Time
  - 1.8.1.1 Selection Sort
  - 1.8.1.2 Insertion Sort
  - 1.8.1.3 Heap Sort
- 1.8.2 Sorting in Linear Time
  - 1.8.2.1 Counting Sort
  - 1.8.2.2 Radix Sort
  - 1.8.2.3 Bucket Sort
  - 1.8.2.4 Bubble Sort

1.9 Searching Algorithm
- 1.9.1 Linear/Sequential Searching Algorithm
- 1.9.2 Binary Search
- 1.9.3 Ternary Search

Objectives ...

After learning this chapter you will be able:

- To write step by step Algorithm and Programs.
- To do comparative study of Algorithms and execution for the same problem.
- To learn complexity measures for Algorithms.
- To do a comparative study of different Sorting and Searching Techniques for the same set of elements.

1.1 INTRODUCTION

- **Algorithm** is "a process or set of rules which to follow in calculations or for solving problem and operations".
- It is a set of rules/instructions that step-by-step defines how work or task is to be executed to return expected results.
- Various criteria considered for designing algorithm.
- For any program to write in any programming language, we require planning and which can be done using algorithm.

1.1.1 Algorithm Definition and Characteristics

What is Algorithm?

- Computer Algorithm is step by step method to solve a problem by using computer.
- **Definition:** An Algorithm is a finite set of instructions that if followed, accomplishes a particular task.

Criteria of Algorithm:

- Algorithm must satisfy following criteria:
  - **(i) Input:** An algorithm should have zero or more entries are supplied as external.
  - **(ii) Output:** An Algorithm should generate at least one output.
  - **(iii) Finiteness:** There should be finite number of instructions.
  - **(iv) Definiteness**: Every step of an algorithm must be precisely defined. i.e. It must be unambiguous and clear .
  - **(v) Effectiveness:** Each instruction should be basic i.e. it can be done by manually, i.e. by using paper and pencil.

1.1.2 Analyzing Algorithm

1. Creating Algorithm:

- It is an art to design an algorithm. It can be created using following techniques:
  1. Divide and Conquer
  2. Greedy Method
  3. Backtracking
  4. Dynamic Programming and so on.

2. **Expressing Algorithm**:

- Usually algorithm is expressed in Assembly language statements. For example, ALGOL 60.

3. **Validating Algorithm:**

- Once algorithm is created it is needed to check that whether it computes output for illegal input also.

4. **Testing of Algorithm**:

(i) Debugging: Algorithm executed on sample data and results are compared with expected output.

(ii) Profiling: It means executing correct program on sample data and measuring its time and space complexity.

1.1.3 Algorithm Specification (Pseudo Code Convention)

- Block of statements should be enclosed within curly braces. For example, { }
- Semicolon should be at end of each statement.
- Comment can be like //
- Data type need not be declared.
- Data types if used same as in basic C language.
- node or structure can be declared as,

```
node:= record
{ datatype1 var1;
 datatype2 var2;
….
node * ptr;
};
```

- Assignment operator is like :=
- Boolean values are true or false.
- Relational and other operators are same as C language.
- Control structures:

1.
```
if condition then
    //statements
else
    //statements
```

2.
```
while condition do
{
    //statements
}
```

3.
```
for i:=1 to n step 1 do
//same as do-while
{
   //statements
}
```
4.
```
repeat
{
   //statements
}
until (condition);
```
[**Note:** Here, condition can be enclosed in( ) brackets]

- Input can be denoted as:

  `read a;`

  OR

  Take as argument to algorithm (i.e. considered as passed direct argument value to algorithm) For example, `Algorithm Algo(a){ }`
- Output can be denoted as:

  `write a;`

  OR

  `print a;`

  OR

  return value to algorithm by return statement. For example, `return a;`

1.1.4 Algorithm Types

- Algorithm can be categorized into:
  1. Simple algorithm
  2. Iterative algorithm
  3. Recursive algorithm

1. Simple algorithm:

- This type of algorithm does not contain any iterative statement or loop or recursive statement.

Example1: Algorithm to find maximum of two numbers.

```
Algorithm Max(n1,n2)
{  //to find maximum of 2 numbers
   if(n1>n2) then
         return n1;
   else
         return n2;
}
```

2. Iterative/Non-recursive algorithm:

- This algorithm uses **Loops** in it. For example, Algorithm may contain do-while, for, while or any other iterative construct.

Example 2: Algorithm to find factorial of a number. For example, $3! = 3 \times 2 \times 1 = 6$.

```
Algorithm fact(num)
{
//to find factorial of given num
if (num=0) or (num=1) then
 //no need for ==sign
 return 1;
else
 return (num * fact(num-1));
}
```

Tracing

```
fact(3)
  ⇒3!=0
  ⇒return (3*fact(2))
  ⇒2!=0
  ⇒return(2*fact(1))
  ⇒1!=0
  ⇒return(1*fact(0))
  ⇒0!=0
return 1
```

Output: $= 3 \times 2 \times 1 = 6$

3. Recursive Algorithm:

- This algorithm uses recursive statement i.e. calls to the same algorithm.

(i) Direct Recursive Algorithm:

Here, Algorithm calls itself.

```
Algorithm fact(num)
{
   if num=0 or num=1 then
   return 1;
   else
   return (num*fact(num-1));
}
```

(ii) Indirect Recursive Algorithm:

Algorithm calls recursively to same algorithm with the help of another algorithm. i.e. here, X() calls itself via Y().

```
Algorithm X()          Algorithm Y()
{                      {
--                     --
Y();                   X();
--                     --
--                     --
}                      }
```

1.1.5 Examples of Algorithm

Example 1: Write an algorithm for finding maximum element from the array of n elements.

C Program

```
void maximum(arr[ ],n)
{
max=arr[0];
for (i=0;i<=n;i++)
{
if(max<arr[i])
max=arr[i];
}
printf(max);
}
```

Algorithm

```
Algorithm maximum(arr[ ],n)
{
max=arr[1]; //1st element at 1st index not 0th index
for i:=1 to n do
{
 if(max<arr[i]) then
 max=arr[i];
}
return max;
}
```

Tracing:

2	4	6	3	9

max = arr[1]=2
i=2 2 < 4 =>max=4
i=3 4 < 6 =>max=6
i=4 6 < 3 =>no max changed, so just i++
i=5 6 < 9 =>max=9
print 9

Example 2: Write an algorithm to print Fibonacci series: 0 1 1 2 3 .....

```
Algorithm FibSeries(last)
{
   X=0;
   Y=1;
   write X;
   write Y;
   for i:=1 to last-2 do
   {
      Z:= X+Y;
      write Z;
      X:=Y;
      Y:=Z;
   }
}
```

Example 3: Write an algorithm to find GCD of 2 numbers.

```
Algorithm GCD(a,b)
{
   if (a=b) then
   return a;
   if (a>b) then
   return GCD(a-b,b);  //a is big
   else
   return GCD(a,b-a);  //b is big
}
```

Example 4: Write an algorithm to check prime number.

```
Algorithm Prime (n)
{
   flag := 0;
   if (n:=2) then
   return " Number is prime";
   for i:=2 to n/2 do
   {
      if (n % i=0) then      //if we get any divisor
      {
         flag =1;
         break;
         else
         flag:=0;
      }
   }
   if (flag:=0) then
   write " Number is prime"
   else
   write "Number is not prime";
}
```

1.2 COMPLEXITY ANALYSIS: SPACE COMPLEXITY

- The better the time complexity of an algorithm is the faster the algorithm performs task. Apart from time complexity, its space complexity is also important. This is basically the number of memory locations which an algorithm requires.
- A good algorithm keeps this number very small possible number. Some time, Space Complexity word is used as Auxiliary Space.

- Following are the correct definitions of Auxiliary Space and Space Complexity:
  - **Auxiliary Space** is nothing but the extra space or temporary space used by an algorithm for doing calculations.
  - **Space Complexity** of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity considers both Auxiliary space as well as space used by input.
- Space complexity can be calculated as **S(p) = C + Sp**
  - Where, `C` is fixed part which is independent on instance characteristics. For example, space required for algorithm itself, space for linking, loading addresses.
  - `Sp` is variable part that depends on space required for variables declared in an algorithm.

Example 1:

```
Algorithm S( )
{
   writeln "welcome";  // As here no variable is declared S(P) = 0
}
```

Example 2:

```
Algorithm Sum(a,n)
{
   for i=1 to n do
      print (a[i]);
}
```

S(p)= n+2

- As array contains n locations, n may vary. So, n locations of storage required for array, one location for storing i, one location for value of n.

Example 3:

```
Algorithm Sum(a,n)
{
   if(n=1) then
      return (a[n]);
   else
      return (a[n] + Sum(a,n-1));
}
```

S(p)=n*(n+1)

- As n locations vary in array. If n=3 then array of 3 locations created. If n=5 then 5 locations in array created and so on. So n space required for array, one location for storing value of n, i.e. (n+1) and as function is recursively called n times different (n+1) locations created.

1.3 COMPLEXITY ANALYSIS: TIME COMPLEXITY

- **Time Complexity:** It is amount of time taken by program for execution i.e. running time of program. Here, total number of times statement is executed. It is also called frequency count. Calculation of exact time may be difficult. But rough estimation can be possible. Execution time is proportional to frequency count.
- **Time Complexity depends on fixed part (C) and variable part (Tp).**
  - Here, fixed part `C` is the time required for linking and loading time.
  - variable part `(Tp)` is thee total time that varies according to number of times statement gets executed.

    T(p) = C + Tp
- In the following examples, algorithms calculate the number of times statement executes.

Example 1:

```
Algorithm Sum (a,m)
{   s=0;            //cnt +1 (assume cnt=0 so cnt=0+1=1) = 1
    for i=1 to m do //cnt increments m+1 times (1 time for failure of
                    condition in loop) = m+1
    s=s+a[i];       //cnt increments for exactly m times = m
    return s;       //cnt +1 = 1
}
```

T(p) = 1 + (m + 1) + m + 1 = 2m + 3

Example 2:

Note: In following table, s/e:The number of steps per execution (s/e) of each statement.

Statements	s/e	Frequency count	Total = s/e* Frequency count
`Algorithm X()`	0	0	0
`{`	0	0	0
`Oldv=1;`	1	1	1
`Newv=1;`	1	1	1
`Print Oldv,Newv;`	1	1	1
`n=20;`	1	1	1
`for i=1 to n do`	1	n+1	n+1
`a=Oldv+Newv;`	1	n	n
`print a;`	1	n	n
`Oldv=Newv;`	1	n	n
`Newv=a;`	1	n	n
`end for;`	0	0	0
`}`	0	0	0
			Total: 5n + 5

If we put n = 20, $5 \times 20 + 5 = 105$.

Example 3:

Statements	s/e	Frequency count	Total
`Algorithm Y()`	0	0	0
`{`	0	0	0
`L=0;`	1	1	1
`for i=1 to n do`	1	n+1	n+1
`for j=1 to n do`	1	n(n+1)	n^2+n
`for k=1 to n do`	1	n(n(n+1))	n^3+n^2
`L=L+i;`	1	n*n*n	n^3
`}`	0	0	0
			Total: $2n^3+2n^2+2n+2$

Example 4:

Statements	s/e	Frequency count	Total
`Algorithm MatrixAdd()`	0	0	0
`{`	0	0	0
`for i=1 to m do`	1	m+1	m+1
`{`	0	0	0
`for j=1 to n do`	1	m(n+1)	m*n+m
`{`	0	0	0
`C[i,j]:=a[i,j]+B[i,j];`	1	m*n	m*n
`}`	0	0	0
`}`	0	0	0
`}`	0	0	0
			Total: 2*m*n+2*m+1

1.4 STEP COUNT: ANALYSIS OF COMPLEXITY

- In the step-count method, we try to calculate the time spent in all parts of the algorithm. The amount of computing represented by one step may be different from that represented by another.

1. Best Case Complexity/Best Step Count:

- It is state that has count is **minimum** number of steps that can be executed for given parameters.

- if $X_1, X_2, ..., X_n$ be the elements time count can be calculated as:

  T_n = **min**($TX_1, TX_2, ..., TX_n$).

2. Worst Case Complexity/Worst Step Count:

- It is state that has count is **maximum** number of steps that can be executed for given parameters.
- This is slowest time algorithm. If $X_1, X_2, ..., X_n$ be the elements time count can be calculated as:

  T_n = **max**($TX_1, TX_2, ..., TX_n$).

3. Average Case Complexity/Average Step Count:

- It is state that has count is **Average** number of steps that can be executed for given parameters.

  $T_n = \sum(TX_1, TX_2, ..., TX_n)/n$. Where, n is total number of inputs.

1.5 ASYMPTOTIC NOTATION

- Asymptotic notation is used to make comparisons between algorithms easily. It is mostly preferred to describe execution time of algorithm.
- Asymptotic notations do not focus on exact value of time or space complexity but focuses on lower and upper bounds of time and space complexity.

1.5.1 Types of Asymptotic Notations

- Three types of asymptotic notations are:
  1. Big Oh(O)
  2. Big Omega (Ω)
  3. Big Theta (θ)

1.5.1.1 Big O(Big Oh Notation) (called Upper Bound)

- The most common notation used is "big O" notation. For example, $n^2 + 3n - 4 = O(n^2)$ (read as "big oh of n squared"). This means, intuitively, the important part of $n^2 + 3n - 4$ is the n^2 part. The expression O is also called **Landau's symbol**.
- **Definition**: Consider f(n) and g(n) be two functions, in which n is a positive integer. We write f(n) = O(g(n)) if and only if there exists a real number c and positive integer n_0 satisfying f(n) <= c*g(n) for all, n >= n_0. (And we say, "f of n is big oh of g of n.")
- We might also say or write f(n) is in O(g(n)), because we can think of O as a set of functions all with the same property.
- For example, the functions like $n^2 + n$, $4n^2 - n \log n + 12$, $n^2/5 - 100n$, $n \log n$, $50n$, and so forth are all $O(n^2)$ i.e. greater than n^2. Every function f(n) bounded above by some constant multiple g(n) for all values of n greater than a certain value is O(g(n)).

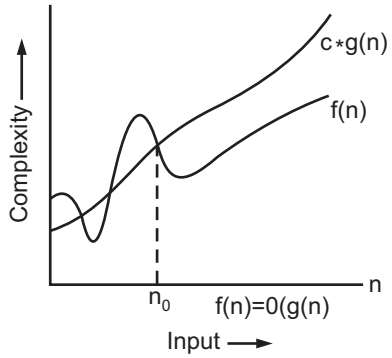

Fig. 1.1: Big Oh notation - Upper bound of a function

Examples:

Example 1: Show that $3n^2 + 4n - 2 = O(n^2)$.

Solution: We need to find c and n_0 such that:

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

- Divide both sides by n^2, getting:

$$3 + \frac{4}{n} - \frac{2}{n^2} <= c \text{ for all } n>= n_0.$$

- If we choose $n_0 = 1$, then we needed a value of c such that:

$3 + 4 - 2 <= c$

- We can set c equal to 6. Now we have:

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

Example 2: $f(n) = 2n^2 + 3n$ and $g(n) = 3n^2 + 2n$, Prove that, f(n) = O(g(n)).

Solution: Here, first consider values of n from 1, 2, 3 ... so on, at least 5 to 6 values. We have to find f(n) and g(n) as per substituting by n value.

n	$f(n) = 2n^2 + 3n$	$g(n) = 3n^2 + 2n$
1	5	5
2	14	16
3	27	33
4	44	56
5	...	...
6	...	...
7	...	...
8	152	800

- Compare f(n) and g(n). Here f(n)<=g(n) for n>=1. No need for multiplying g(n) with constant so c=1 from formula, f(n)<=c*g(n). So, f(n) = O(g(n)).

1.5.1.2 Big Omega Notation Ω (Called Lower Bound)

- **Big Omega** in which f(n) is said to be Ω (g(n)), if ∃ a positive real constant c and a positive integer n_0 such that f(n) >= c*g(n) .
- We define big-oh notation by saying f(n)=O(g(n)), if there exists constant named c so that for all n, f(n) ≤ c g(n). If the same followed for all c > 0, then f(n) = o(g(n)), becomes the little-oh notation.
- Big-oh and little-oh notation mostly used terms in analyzing algorithms because we can ignore implementation issues that could cost a constant factor.
- For lower bounds, we use the big-omega notation that shown as, f(n) = Ω(g(n)) and which is defined by some constant c > 0 and all large possible n, f(n) ≥c g(n). This follows symmetry property, f(n) = O(g(n)) if and only if g(n) = Ω(f(n)). But it is not corresponding to how we actually prove lower bounds.

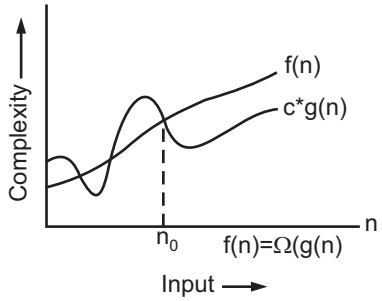

Fig. 1.2: Big Omega Notation (lower bound) - Ω(g(n))

Example 1: f(n) = n^2 + 15 and g(n) = $2n^2$, Prove that, f(n) = Ω(g(n)) or not.

n	f(n) = n^2+15	g(n) = $2n^2$
1	16	2
2	19	8
3	24	18
4	31	32
8	79	128

- Here, first consider values of n from 1,2,3...so on, at least 5 to 6 values.
- Calculate f(n) and g(n) as per value substituting by n.
- Compare f(n) and g(n). Here, f(n)!>=g(n).f(n)!= Ω(g(n))

1.5.1.3 Big Theta notation: θ (Asymptotic Tight Bound)

Definition:

- **Big Theta**: A function f(n) is θ g(n)) iff ∃ two positive constants, c_1 and c_2 and a positive integer n_0, such that c_1*g(n) <=f(n) <= c_2*g(n) condition followed.

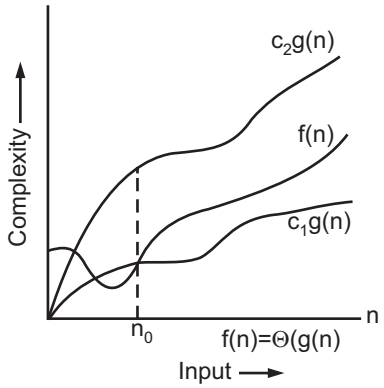

Fig. 1.3: Big Theta Notation

Example 1: f(n) = n^2 + 3n and g(n) = $2n^2$ + 2n, Prove that, f(n) = θ(g(n)).

n	Consider c_1 = 1 g(n) = $2n^2$+2n	f(n) = n^2+3n	Consider c_2 = 1 g(n) = $2n^2$+2n
1	4	4	4
2	12 !<=10	10	12
3	24	18	24
4	40	28	40

- Here, first consider values of n from 1, 2, 3 ... so on, at least 5 to 6 values.
- Calculate f(n) and g(n) as per value substituting by n, and considering c_1, c_2 = 1.
- Compare f(n) and g(n). Here no c_1*g(n) <=f(n) <= c_2*g(n) condition satisfied. So, again considering c_1 = 1, c_2 = 2. Let's do it again here, except this time no condition is satisfied so, f(n)! = θ(g(n)).

1.5.2 Rank Ordering

Example:

Rank the following functions in their increasing order of growth rates.
n^2, n^4 + 2n, n^3, n^2 – 18n, ... So on.

Solution:

- Here, you have to calculate values of functions by putting value of n as above by assumption.
- Arrange functions in ascending order.
- For example, above functions rank ordering or growth rate is,

$n^2 \le n^2 - 18n \le n^3 \le n^4 + 2n$

n	n^2	n^4 + 2n	n^3	n^2 – 18n
1	1	3	1	– 17
2	4	20	8	– 32
4	16	264	64	– 56
6	36	1308	216	– 72

1.6 RECURSIVE ALGORITHMS

- As we already learned recursive algorithm and its types in 1.1.4. Here we learn one of most important recursive algorithm.

1.6.1 Tower of Hanoi

- It is type of recursive algorithm. Consider 3 towers are given as A, B, C.
- A is source tower containing number of disks arranged in decreasing order. We have to place n disks from tower A to destination tower B with the help of intermediate tower C. On each tower disks should be placed in descending order.

```
Algorithm TOH(n, A, B, C)
{ //move n disks from tower A to B using C
   if n=1 then
   write "move disk n from",A,"to",B;
   else
   {
      TOH((n-1),A,C,B);
      write "move topmost disk from ",A," to", C;
      TOH((n-1),C,B,A);
   }
}
```

- **Example for 1 disk and 3 towers:**

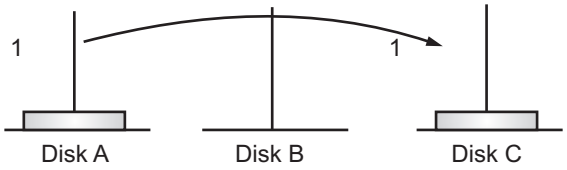

Fig. 1.4: For 1 disk and 3 towers

Or direct on Disk B.

- **Example for 2 disks and 3 towers:**

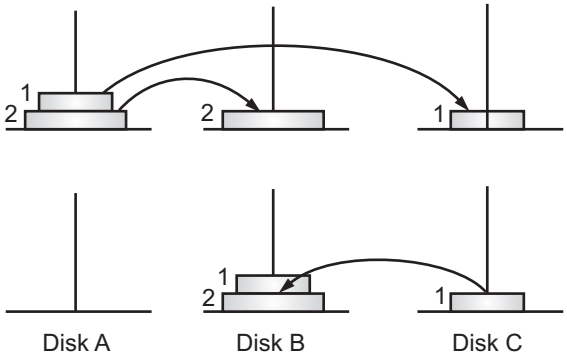

Fig. 1.5: For 2 disks and 3 towers

- **Example for 3 disks and 3 towers:**

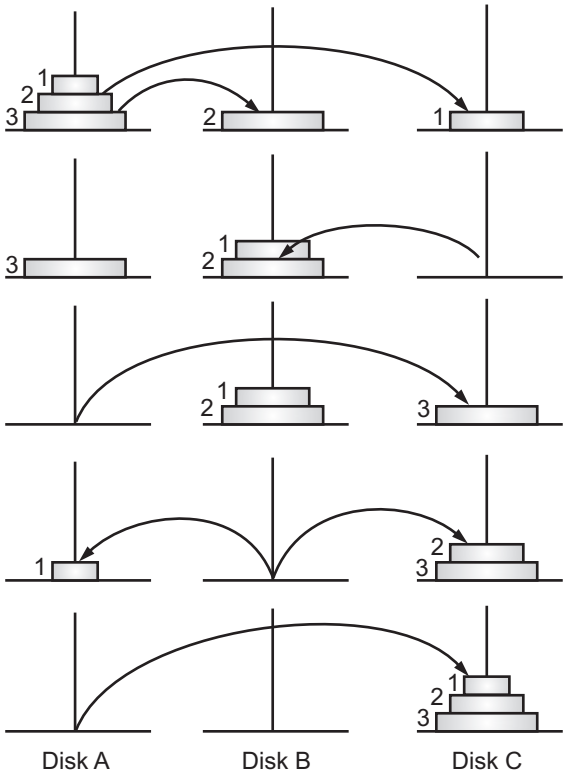

Fig. 1.6: For 3 disks and 3 towers

1.6.2 Recursion Types

(i) Head recursion: If algorithm contains recursive call as a first statement of algorithm then it is having head recursion.

Example: Power (x, y) i.e. x^y = x*x*x*....y times

```
Algorithm power(x, y)
{
if(y>1) then
return (x*power(x,(y-1)));
else
return x;
}
```

Tracing:

```
Power(2,3) i.e. 23
    3>1=>2 * power(2,2)
    2>1=>2 * 2 *power(2,1)
    1>=1 =>2 * 2 * 2=>8
```

(ii) Tail recursion: If algorithm contains recursive call as a last statement of algorithm then it is having tail recursion.