RV College of Engineering®
Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi
Approved by AICTE New Delhi

Introduction to Algorithms

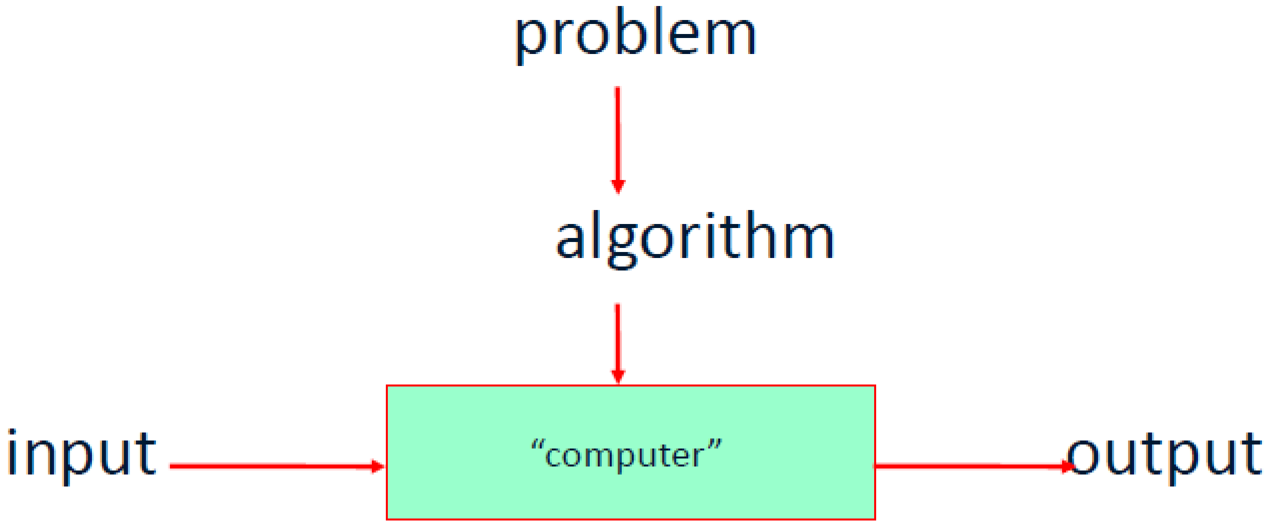
UNIT 1

Prof. Rajesh R M
Dept. of AIML
RV College of Engineering
Bengaluru

Go, Change the World

Introduction to Algorithms

- **A sequence of unambiguous instructions for solving a problem, i.e. for obtaining the required output for any legitimate input in a finite amount of time**



Introduction to Algorithms

- **"algos" = Greek word for pain.**
- **"algor" = Latin word for to be cold.**

Why study this subject?

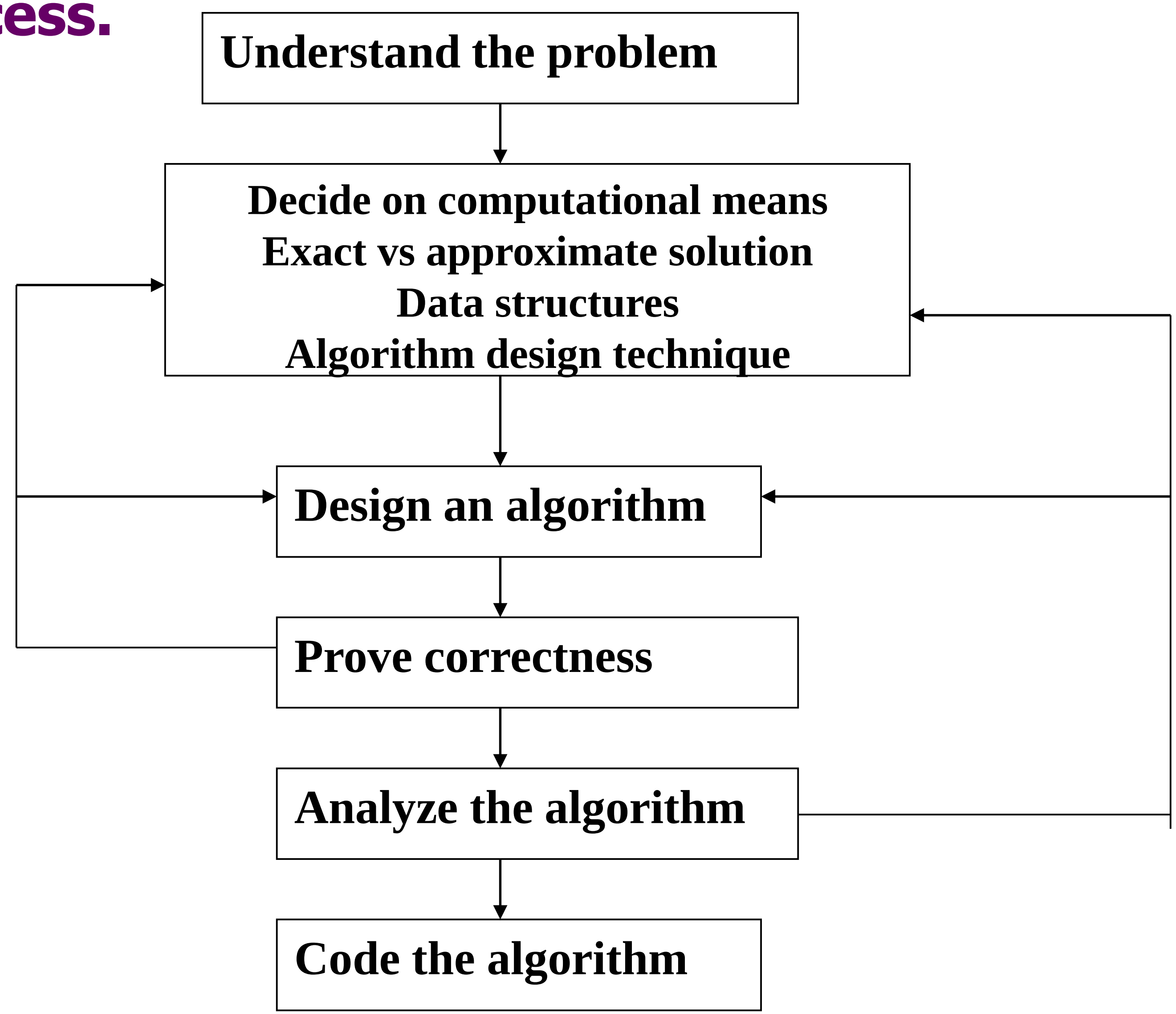
- Efficient algorithms lead to efficient programs.
- Efficient programs sell better.
- Efficient programs make better use of hardware.
- Programmers who write efficient programs are preferred.

PROPERTIES OF AN ALGORITHM

- An algorithm takes zero or more inputs
- An algorithm results in one or more outputs
- All operations can be carried out in a finite amount of time
- An algorithm should be efficient and flexible
- It should use less memory space as much as possible
- An algorithm must terminate after a finite number of steps.
- Each step in the algorithm must be easily understood for some reading it
- An algorithm should be concise and compact to facilitate verification of their correctness.

Introduction to Algorithms

Algorithm design and analysis process.



Introduction to Algorithms

What does it mean to understand the problem?

- What are the problem objects?
- What are the operations applied to the objects?

Deciding on computational means

- How the objects would be represented?
- How the operations would be implemented?
- Build a computational model of the solving process

Prove correctness

- Correct output for every legitimate input in finite time
- Based on correct math formula

Introduction to Algorithms

Efficiency: time and space

Simplicity: Keep it simple and easily understandable to others

Generality: range of inputs, special cases

Optimality: no other algorithm can do better

Coding

How the objects and operations in the algorithm are represented in the chosen programming language?

Important problem types

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

Fundamental data structures

Linear data structures

- Array
- Linked list
- Stack
- Queue

Operations: search, delete, insert

Implementation: static, dynamic

Non-linear data structures

- Graphs
- Trees : connected graph without cycles
  - Rooted trees
    - Ordered trees
      - Binary trees

Graph representation: **adjacency lists,**
adjacency matrix

Tree representation: **as graphs; binary nodes**

Introduction to Algorithms

→GCD of 2 numbers Algorithm

Euclid's algorithm for computing gcd*(m, n)*

Step 1 If *n* = 0, return the value of *m* as the answer and stop; otherwise,
proceed to Step 2.

Step 2 Divide *m* by *n* and assign the value of the remainder to *r*.

Step 3 Assign the value of *n* to *m* and the value of *r* to *n*. Go to Step 1.

Introduction to Algorithms

→GCD of 2 numbers using Euclidean Algorithm

ALGORITHM *Euclid(m, n)*
//Computes gcd*(m, n)* by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers *m* and *n*
//Output: Greatest common divisor of *m* and *n*
while *n* = 0 **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
    return *m*

Introduction to Algorithms

Example : Gcd (33,12)

→GCD of 2 numbers using Euclidean Algorithm

ALGORITHM *Euclid(m, n)*
while *n* = 0 **do**
  r ←*m* mod *n*
  m←*n*
  n←*r*
  return *m*

Q	M	N	R
2	33	12	9
1	12	9	3
0	9	3	0
	3	0	

Introduction to Algorithms

→GCD of 2 numbers using Consecutive Integer Checking Method

Consecutive integer checking algorithm for computing gcd*(m, n)*
Step 1 Assign the value of min{*m, n*} to *t.*
Step 2 Divide *m* by *t.* If the remainder of this division is 0, go to Step 3;
otherwise, go to Step 4.
Step 3 Divide *n* by *t.* If the remainder of this division is 0, return the value of *t* as the answer and stop;
otherwise, proceed to Step 4.
Step 4 Decrease the value of *t* by 1. Go to Step 2.

Introduction to Algorithms

→**GCD of 2 numbers using Consecutive Integer Checking Method**

Example:

gcd(9,3)

STEPS FOR WRITING AN ALGORITHM

An algorithm consists of two parts.

- The first part is a paragraph, which tells the purpose of the algorithm, which identifies the variables, occurs in the algorithm and the lists of input data.
- The second part consists of the list of steps that is to be executed.

STEPS FOR WRITING AN ALGORITHM

Step 1: Identifying Number

Each algorithm is assigned an identifying number.
Example: Algorithm 1. Algorithm 2 etc.,

Step 2: Comment

Each step may contain comment brackets, which identifies or indicates the main purpose of the step.

The Comment will usually appear at the beginning or end of the step. It is usually indicated with two square brackets [ ].

- **Example :**

  Step 1: [Initialize]

  set K : = 1

STEPS FOR WRITING AN ALGORITHM

Step 3 : Variable Names
It uses capital letters. Example MAX, DATA. Single letter names of variables used as counters or subscripts.

Step 4 : Assignment Statement
It uses the dot equal notation (: =). Some text uses or or = notations

Step 5 : Input and Output
Data may be input and assigned to variables by means of a Read statement. Its syntax is :
READ : variable names
Example:
READ: a,b,c

Similarly, messages placed in quotation marks, and data in variables may be output by means of a write or print statement. Its syntax is:
WRITE : Messages and / or variable names.
Example:
Write a b c

STEPS FOR WRITING AN ALGORITHM

Step 7 : Controls:

It has three types

(i) : Sequential Logic :

It is executed by means of numbered steps or by the order in which the modules are written

(ii) : Selection or Conditional Logic

It is used to select only one of several alternative modules.

The end of structure is usually indicated by the statement.

[End - of-IF structure]

The selection logic consists of three types

Single Alternative,Double Alternative and Multiple Alternative

Performance Analysis or Efficiency of the Algorithm

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

- Space Complexity
- Time Complexity

1. Space Complexity

- Space required completing the task of that algorithm.
- It includes program space and data space

S(p)=C + Sp

S(p) – Space Complexity of a problem

C – Constants (Static Memory)(local Variables)

Sp- Variables Part (Dynamic Memory) (Axillary Variables)

Efficiency of the Algorithm

2. Time Complexity

Time required to complete the task of that algorithm.
Measured in seconds, nano seconds and mili seconds

Time efficiency depends on

- System in which the program being executed
- Programming Language used
- Type of Compiler Used

Introduction to Algorithms

Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Efficiency of the Algorithm

2. Time Complexity

Time required to complete the task of that algorithm.
Measured in seconds, nano seconds and mili seconds

Time efficiency depends on

- System in which the program being executed
- Programming Language used
- Type of Compiler Used

2. Time Complexity

To find the time complexity of we have formula

T(n)=cop x c(x)

Cop- Time taken for one execution of the basic operation

Function which express how many time the basic operation is been executed

Time efficiency

Best case – Minimum no of times the basic operation gets executed for input n

for Ex: Linear search – 1

Worst case- Maximum no of times the basic operation gets executed for input n

for Ex: Linear search – n

Average case -the basic operation gets executed for input n, between minimum and maximum

Introduction to Algorithms

Asymptotic Notations

To compare and rank the order of growth of a function we use asymptotic notations

5 different types of notations are

1. Big oh O - Denotes <=
2. Big Omega Ω - Denotes >=
3. Theta Θ - Denotes =
4. Small oh o - Denotes <
5. Small omega ω - Denotes >

t(n) and g(n) be two non-negative functions on a set of natural no's

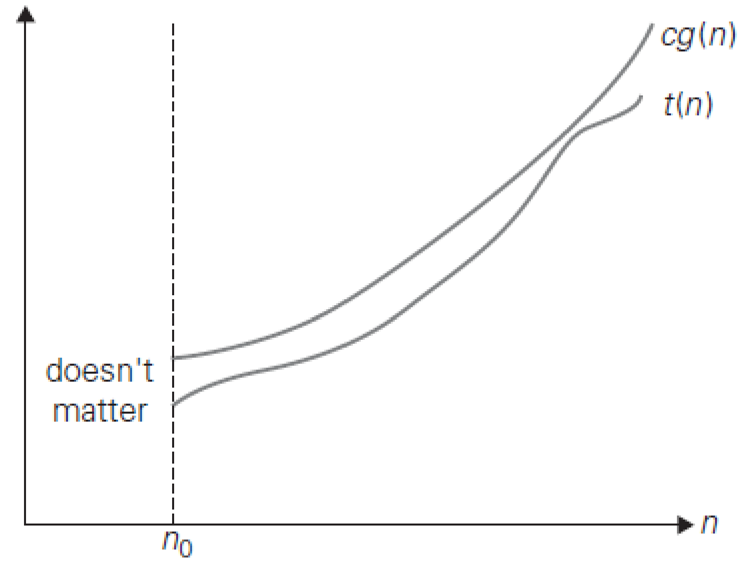
t(n) – actual time taken be an algorithm

g(n)-sample function

Asymptotic Notations
1. Big oh O (Worst Case)

A function t(n) is said to be O(g(n)), denoted by t(n) ϵ O(g(n)), if t(n) is bounded above some constant multiples of g(n) for all large values of n, if there exists a positive integer constant c and positive integer n_0 satisfying the statement

$t(n) \le c\, g(n)\ \forall\ n \ge n_0 \qquad C > 0$ and $n_0 \le 1$



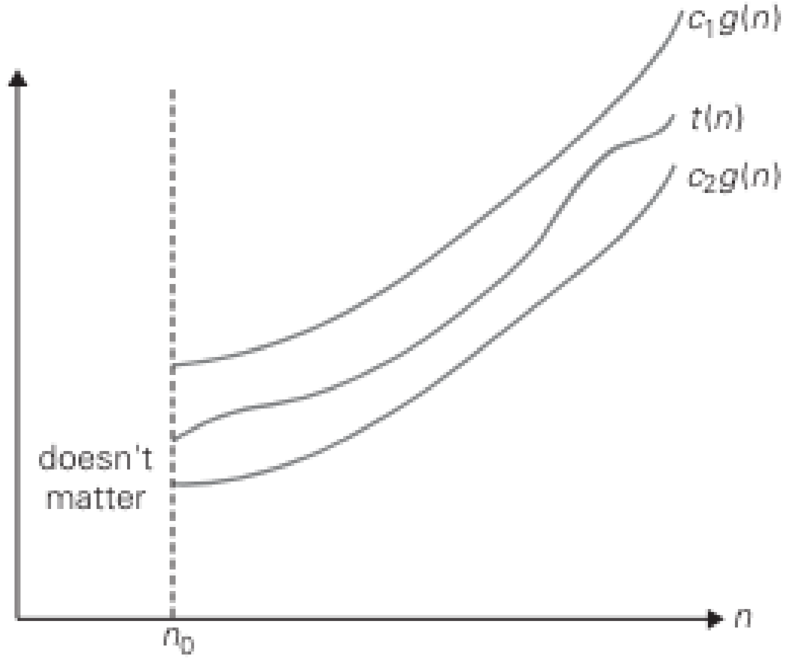
Big-oh notation: $t(n) \in O(g(n))$.

Asymptotic Notations

2. Theta Θ (Average Case)

DEFINITION A function t (n) is said to be in (g(n)), denoted t (n) ∈ (g(n)), if t (n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n0 such that

$$c_2 g(n) \le t(n) \le c_1 g(n) \text{ for all } n \ge n_0.$$



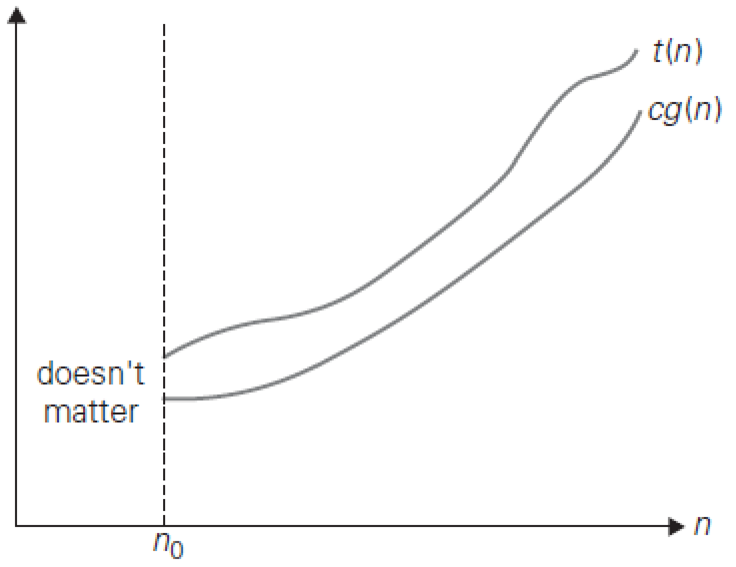
Big-theta notation: $t(n) \in \Theta(g(n))$.

Asymptotic Notations

3. Big Omega Ω (Best Case)

DEFINITION A function $t(n)$ is said to be in $(g(n))$, denoted $t(n) \in (g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer $n0$ such That

$$t(n) \geq cg(n) \text{ for all } n \geq n0.$$



Big-omega notation: $t(n) \in \Omega(g(n))$.

Asymptotic Notations

3. Small Oh

t(n)=o(g(n)) if lim $\lim_{n \to \infty} \frac{t(n)}{g(n)} = 0$

Asymptotic Notations

3. Small Omega

t(n)= ω(g(n)) if $\lim_{n \to \infty} \frac{t(n)}{g(n)} = \infty$

Introduction to Algorithms

Comparison of Efficiency classes using Asymptotic Notations

t(n)=O(g(n)) if $\lim_{n\to\infty} \frac{t(n)}{g(n)} \leq C$

t(n)=Θ (g(n)) if $\lim_{n\to\infty} \frac{t(n)}{g(n)} = C$

t(n)=Ω (g(n)) if $\lim_{n\to\infty} \frac{t(n)}{g(n)} \geq C$

t(n)=o(g(n)) if $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$

t(n)= ω(g(n)) if $\lim_{n\to\infty} \frac{t(n)}{g(n)} \leq \infty$

Using Limits for Comparing Orders of Growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hôpital's rule

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

Using Limits for Comparing Orders of Growth

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n),\\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n),\\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^{3}\end{cases}$$

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2.

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1)\in\Theta(n^2)$. ■

Mathematical Analysis of Recursive Algorithms

General Plan for Analysing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.

2. Identify the algorithm's basic operation.

3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.

4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Mathematical Analysis of Recursive Algorithms

We can analyse the recursive algorithm in 3 ways

1. **Substitution Method (Backword Substitution Method)**
2. **Recursive Tree Method**
3. **Master's Theorem**

Example

```
ALGORITHM F(n)
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0
      return 1
else
      return F(n − 1) ∗ n
```

Mathematical Analysis of Recursive Algorithms Using Backword Substitution

Method

Example

```
ALGORITHM F(n)
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0
      return 1
else
      return F(n − 1) ∗ n
```

1. Identify the input – input n
2. Identify the basic operation – Multiplication – denoted by M
3. Identify the function- let M(n) denotes the no of time the basic operation is executed

Introduction to Algorithms

Mathematical Analysis of Recursive Algorithms Using Backword Substitution Method

1. Identify the input – input n
2. Identify the basic operation – Multiplication – denoted by M
3. Identify the function- let M(n) denotes the no of time the basic operation is executed
4. Identify the base case- if the value of n=0 then there is no basic operation is executed

   i.e. $M(0)=0$

5. Write the recurrence relation $F(n)=F(n-1)+1 \rightarrow M(n)=M(n-1)+1$
6. Using backword substitution method find the order of growth

   $M(n)=\underline{M(n-1)}+1$

Introduction to Algorithms

Mathematical Analysis of Recursive Algorithms Using Backword Substitution Method

6. Using backword substitution method find the order of growth

$M(n)=\underline{M(n-1)}+1$

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

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

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

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

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

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

$M(n)=0+n$

$M(n)=n$

factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120

Mathematical Analysis of Recursive Algorithms Using Backword Substitution Method

```
Algorithm HANOI(src, temp, dest, n)
// Description : Move n disks from source peg to destination peg
// Input : 3 pages, and n disks on source peg
// Output : n disks on destination peg

if n = = 1 then
      Move disk from src to dest
else
      HANOI(src, dest, temp, n – 1)
      HANOI(src, temp, dst, 1)
      HANOI(temp, src, dest, n – 1)
end
```

Mathematical Analysis of Recursive Algorithms Using Backword Substitution Method

For one disk

tower(1,A,B,C) → A->C

for 2 disks

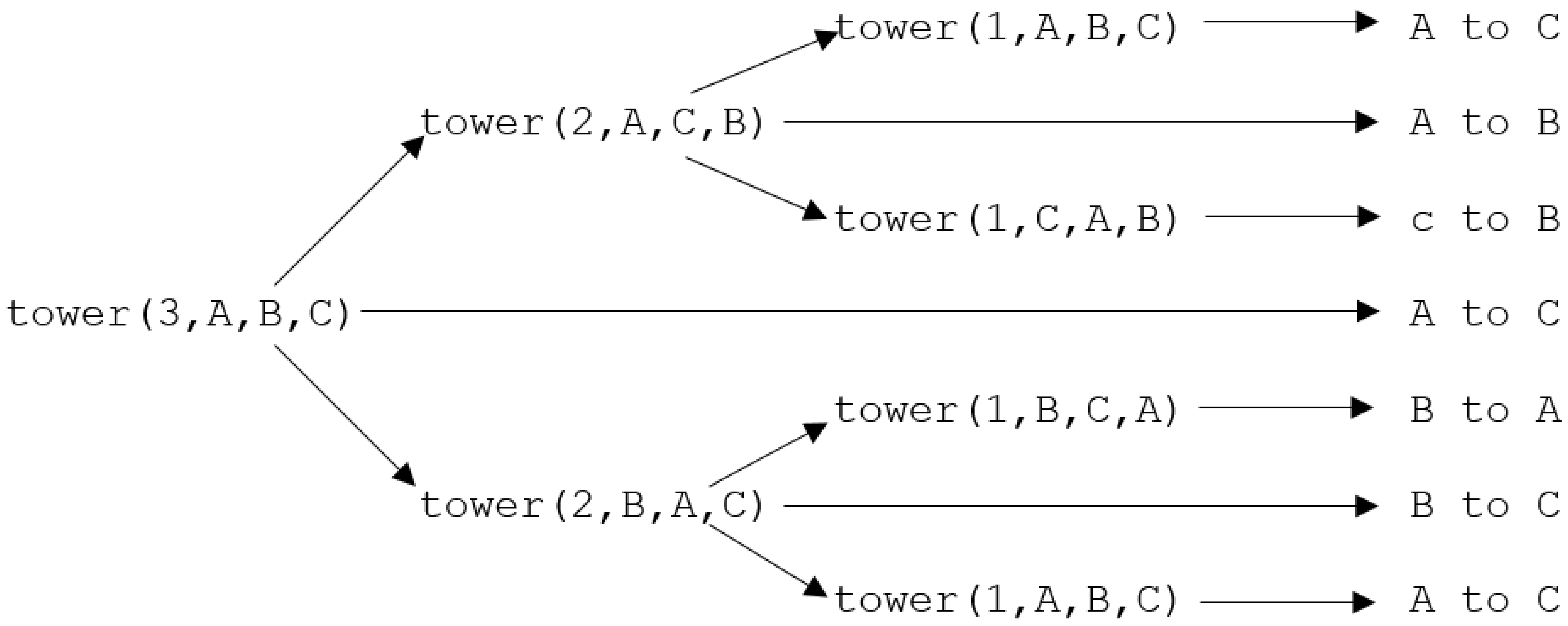
tower(2,A,B,C) → tower(1,A,B,C) → A to C

tower(2,A,B,C) → A to C

tower(2,A,B,C) → tower(1,B<A<C) → B to C

Mathematical Analysis of Recursive Algorithms Using Backword Substitution Method

for 3 disks



Introduction to Algorithms

Mathematical Analysis of Recursive Algorithms Using Backword Substitution Method

1. Identify the input – input n (Number of discs)
2. Identify the basic operation – Movement – denoted by M
3. Identify the function- let M(n) denotes the no of time the basic operation is executed
4. Identify the base case- if the value of n=1 then there is no basic operation is executed

   i.e. $M(1)=1$

5. Write the recurrence relation $F(n)=F(n-1)+1 \ +F(n-1) \rightarrow M(n)=M(n-1)+1+M(n-1)$
6. Using backword substitution method find the order of growth

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

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

Mathematical Analysis of Recursive Algorithms Using Backword Substitution Method

6. Using backword substitution method find the order of growth

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

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

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

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

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

$M(3)=8M(n-3)+4+2+1$

$M(n-k)=2^k M(n-k)+2^{k-1} + 2^{k-2} \ldots\ldots. 2^1 + 2^0$ Since we know that $n = 1$

$= 2^{n-1} M(n-n-1)+2^{n-1-1} + 2^{n-1-2} \ldots\ldots. 2^1 + 2^0$ $M(n-k)=1$

$= 2^{n-1} M(1)+2^{n-2} + 2^{n-3} + 2^{n-4} \ldots\ldots. 2^1 + 2^0$ $k= n-1$

$= 2^{n-1} \cdot 1+2^{n-2} + 2^{n-3} + 2^{n-4} \ldots\ldots. 2^1 + 2^0$ $M(1)=1$

$= 2^n -1 \rightarrow 2^n$

Excersice problem : $x(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$

Introduction to Algorithms

Empirical Analysis of an Algorithms

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.

2. Decide on the efficiency metric *M* to be measured and the measurement unit (an operation count vs. a time unit).

3. Decide on characteristics of the input sample (its range, size, and so on).

4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.

5. Generate a sample of inputs.

6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.

7. Analyse the data obtained.

Introduction to Algorithms

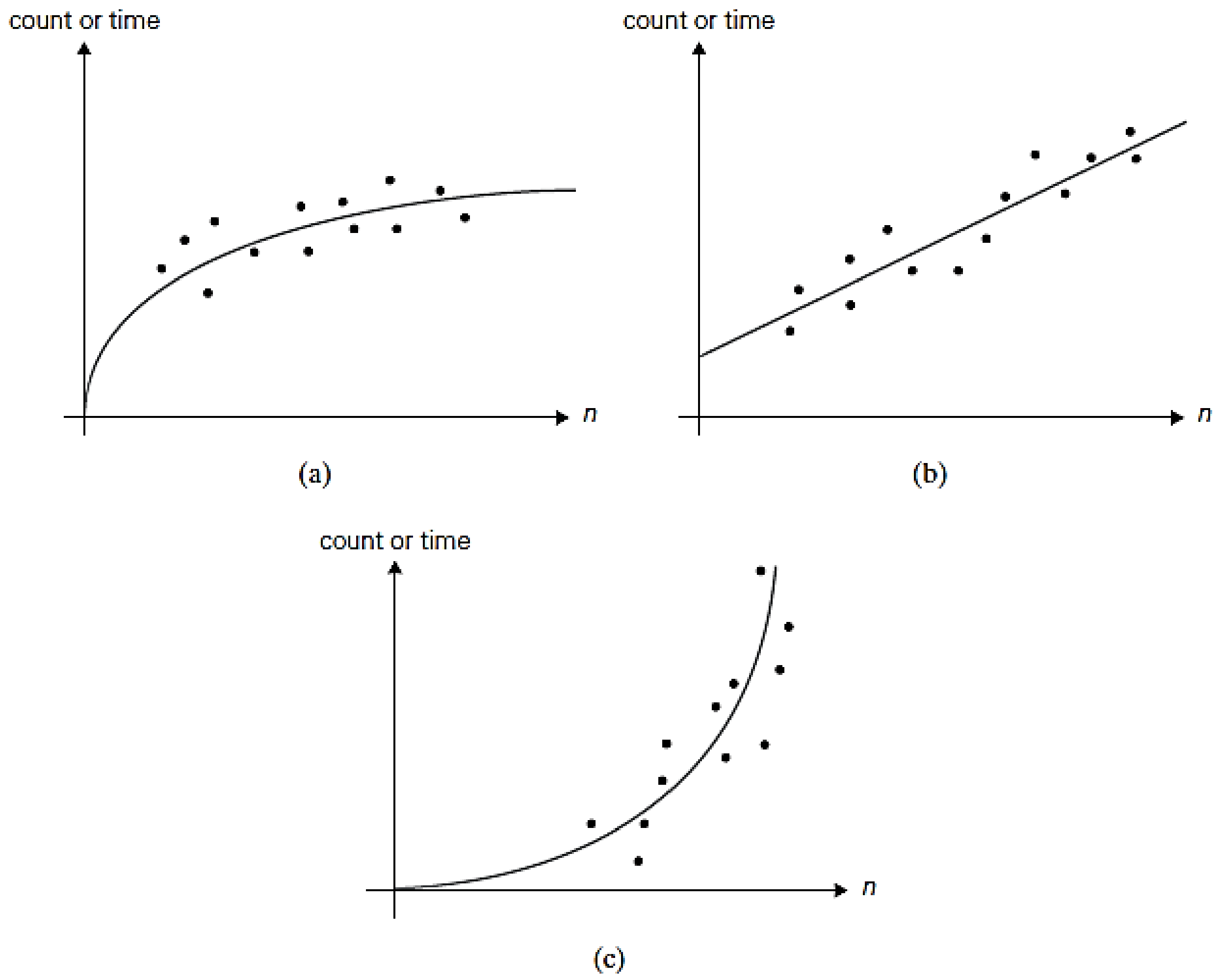
Empirical Analysis of an Algorithms

Empirical Analysis of an Algorithms is denoted below

We need graph to denote the Empirical Analysis of an algorithm.

Where **X** axis Denotes the input size

Y axis denotes the count or Time



Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions.