# DEPARTMENT OF BCA

## ANALYSIS AND DESIGN OF ALGORITHMS

## SUB CODE: 16BCA5D11

## MODULE 1: Role of Algorithms in Computing

# CONTENTS :

**Module 1** : Introduction: What is an Algorithm? Notion of Algorithm, Fundamentals of Algorithmic Problem Solving, Role of algorithms in computing, Algorithms as a technology.

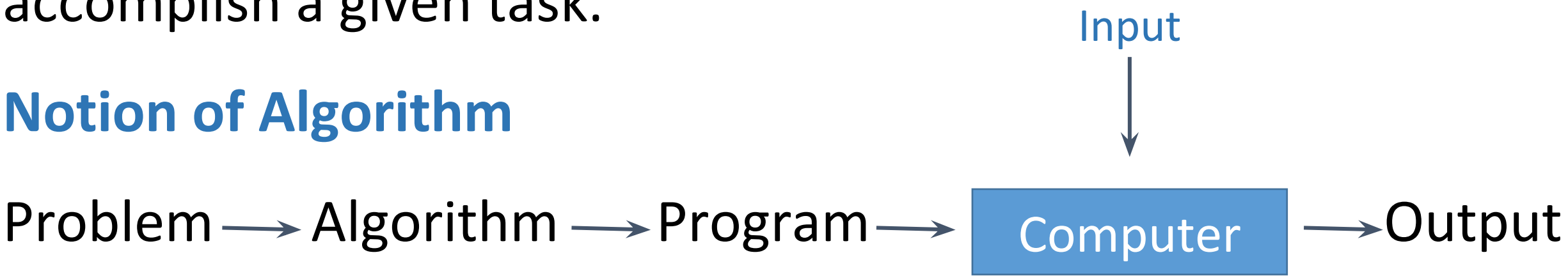
Getting Started: Fundamentals of the Analysis of Algorithm Efficiency, Asymptotic notation and Basic Efficiency Classes, Algorithm design.

**Course Material Link :** https://drive.google.com/file/d/145y7Cu5VGepxzQdIK3GbNlzuX8xbcbY6/view?usp=sharing

# Introduction: What is an Algorithm?

Finite sequence of unambiguous instructions followed to accomplish a given task.

## Notion of Algorithm



# Properties of an Algorithm :

**All algorithms must satisfy the following criteria:**

***Input :*** Zero or more quantities are externally supplied.

***Output :*** At least one quantity is produced.

***Definiteness :*** Each instruction is clear and produced.

***Finiteness :*** If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

***Effectiveness :*** Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

## Advantages of Algorithm

- It is a step-by-step representation of a solution to a given problem, which is very easy to understand.
- It has got a definite procedure, which can be executed within a set period of time.
- It is easy to first develop an algorithm, and then convert it into a flowchart and then into a computer program.
- It is independent of programming language.
- It is easy to debug as every step has got its own logical sequence.

## Disadvantages of Algorithm

- It is time consuming and cumbersome as an algorithm is developed first which is converted into a flowchart and then into a computer program.

# Computing GCD :

## GCD : Greatest Common Divisor

- GCD of two numbers m and n denoted by GCD(m,n)
- Defined as the largest integer that divides both m and n such that the remainder is Zero.
- Defined only for positive integers.

## Different Ways of Computing GCD of two nubers:

1. Euclid's Algorithm
2. Repetative Subtraction
3. Consecutive Integer Checking Algorithm
4. Middle School Procedure using prime factors

# 1. Euclid's Algorithm

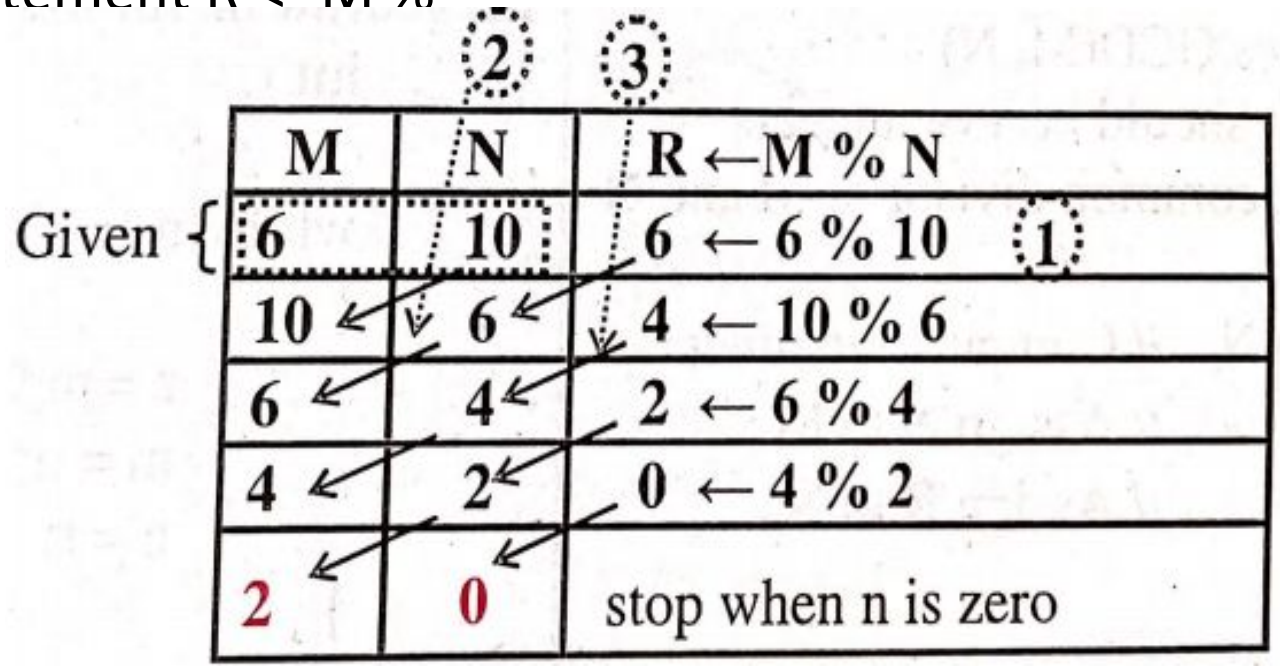
- Named after the mathematician - Euclid, Alexandria

**Procedure :**

Step1 : Compute the remainder using the statement R <- M % N

Step 2: Assign N to M i.e., M <- N

Step 3: Assign remainder R to N i.e., N <- R



## Iterative Algorithm :

Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N

**While** N ≠ 0 do
    R ← M%N
    M ← N
    N ← R
**End While**
Return M

## Recursive Algorithm :

Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N

$$GCD(M,N) = \begin{cases} M & \text{if } N=0 \\ GCD(N, M\%N) & \text{Otherwise} \end{cases}$$

## 2. Consecutive Integer Checking :

- WKT GCD of 2 numbers M and N cannot be more than the samaller of these two numbers.
- To start with : small =min(m,n)
- Divide m and n by small.
- if remainder in both the case is zero, then small will be the GCD.
- Otherwise, decrement small by 1 and repeat the process till both m and n are divisile by small.

| small | m % small | n %small | small is GCD ? |
|---|---|---|---|
| 6 | 10 % 6 = 4 | 6 % 6 = 0 | 6 is not GCD |
| 5 | 10 % 5 = 0 | 6 % 5 = 1 | 5 is not GCD |
| 4 | 10 % 4 = 2 | 6 % 4 = 2 | 4 is not GCD |
| 3 | 10 % 3 = 1 | 6 % 3 = 0 | 3 is not GCD |
| **2** | **10 % 2 = 0** | **6 % 2 = 0** | **2 is GCD** (remainder is 0) |

**Algorithm :**

Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N

```
Small ← min(m,n)
while(1)
      if (m mod small=0)
        if (n mod small =0)
            return small
        end if
      end if
small ← small - 1
end while
return small
```

**Note : This algorithm will not work if one of the input is Zero.**

# 3. Repetative Subtraction :

## Procedure :

- If m is greater than n, perform m-n and store in m
- If n is greater than m, perform n-m and store in n
- Repeat the above 2 steps as long as m and n are not equal
- if ma nd n are same, return m on as GCD.

| M | N | Description |
|---|---|---|
| 10 | 6 | M = 10 – 6 = 4 (Since M > N, subtract N from M and store in M) |
| 4 | 6 | N = 6 – 4 = 2 (Since N > M, subtract M from N and store in N) |
| 4 | 2 | M = 4 – 2 = 2 (Since M >N, subtract N from M and store in M) |
| 2 | 2 | Since M and N are same, the GCD will be either M or N which is 2. |

GCD = 2.

**Algorithm :**

Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N

```
while m ≠ n do
    if (m>n)
        m    m-n
    else
        n    n-m
    end if
end while
return m
```

**Note : If one of two numbers is a Zero. Return non-negative number as the GCD**

# 4. Middle School Procedure:

## Procedure :

Step 1: Find the prime factors of m

Step 2: Find the prime factors of n

Step 3: Identify the common prime factors obtained in step 1 and step 2

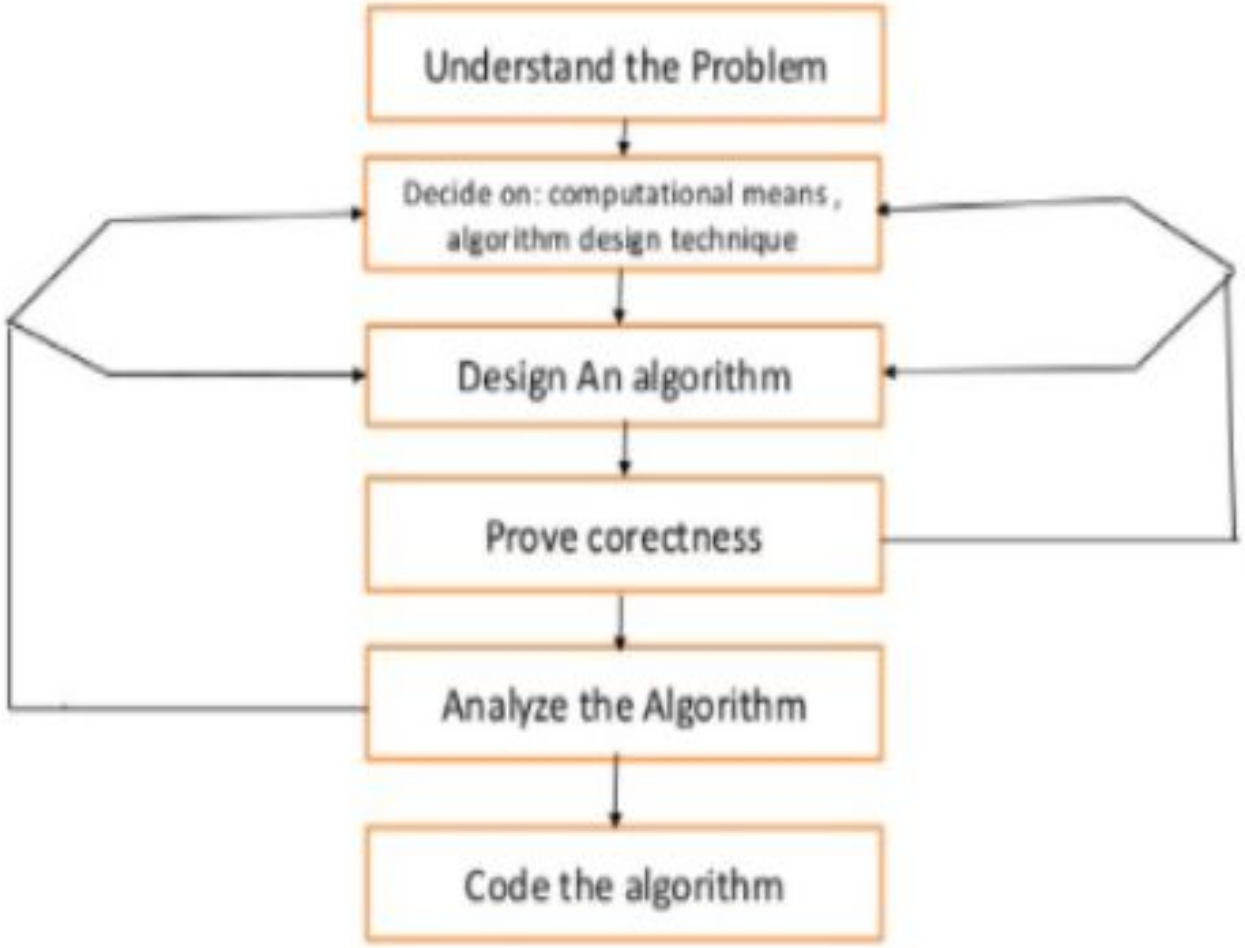
Step 4: Find the product of all common factors and return the result as gcd of two given numbers.

## Disadvantages of Middle School Procedure:

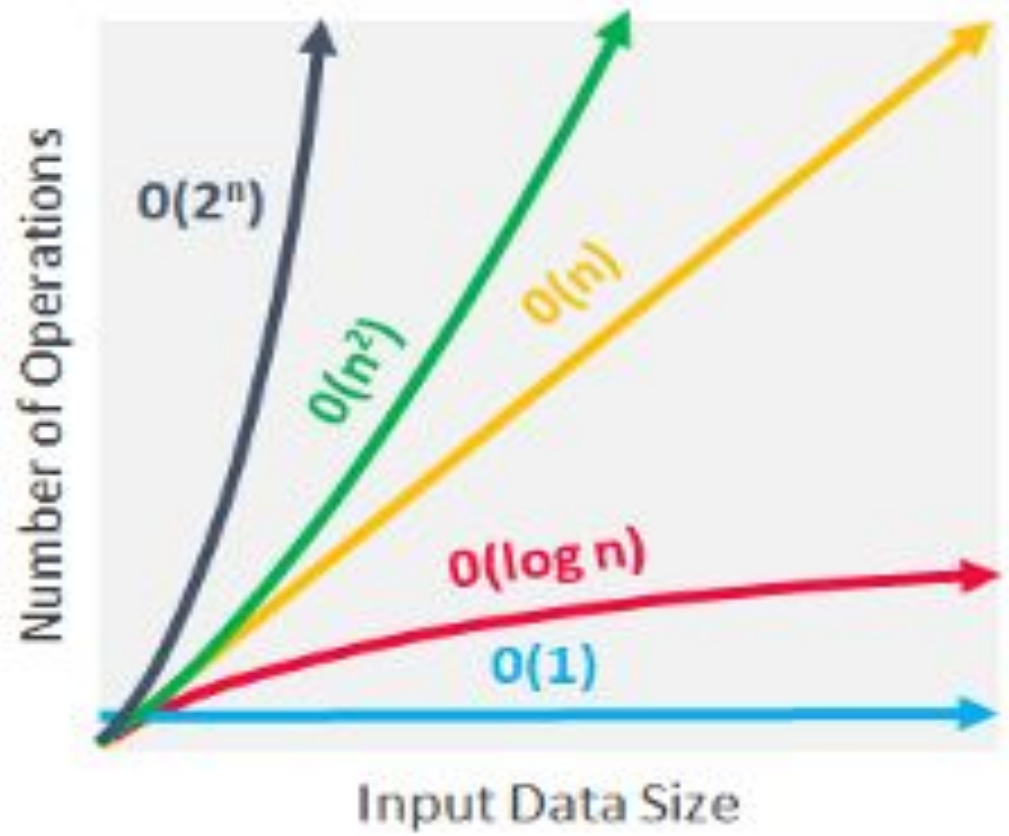
- More Complex
- This algorithm doesnot sepcify how to generate the prime factors and hence it is not a legitimate algorithm.
- This algorithm doesnot sepcify how to find common factors and hence it is not possiblento write the program.

# Fundamentals of Algorithmic Problem Solving:

- Understanding the Problem
- Decision making
- Methods of Specifying an Algorithm
- Proving an Algorithm's Correctness
- Analyzing an Algorithm
- Coding an Algorithm



# aNALYSIS OF ALGORITHMS



# Analysis of Algorithm:

**Purpose :** To design most efficient algorithm.

## On what factors efficiency of algorithm depends?

Efficiency of algorithm depends on 2 factors:

1. Space Efficiency
2. Time Efficiency

# Space Efficiency :

- The space efficiency of an algorithm is the amount of memory required to run the program completely and efficiently.
- If the efficiency is measured with respect to the space then it is called as space complexity.
- Space complexity of an algorithm depends on following factors:
  - **Program space**
  - **Data space**
  - **Stack space**

**Program space:** the space required for storing the machine program generated by the compiler or assembler.

**Data space:** the space required to store the constants variables etc.

**Stack space:** the space required to store the return address along with parameters that are passed to the function local variables etc.

## Time efficiency:

- It is measured purely on how fast a given algorithm is executed.
- The efficiency of an algorithm is measured using time - Time complexity.

## On what factors that time efficiency of an algorithm depends on?

1. Speed of the computer
2. Choice of the programming language
3. Compiler used
4. Choice of the algorithm
5. Size of input or output

# Basic Operation :

- A statement that execute maximum number of times in a function
- It is Analysed by determining the number of time the basic operation is executed

the running time T(n) is given by:

$$T(n) \sim b*C(n)$$

where,

- T is the running time of the algorithm
- n is the size of the input
- b execution time for basic operation
- C represent number of times the basic operation is executed

# Order of Growth :

- We know that some algorithm execute faster for smaller values of n but as the value of n increases they tend to be very slow.
- The behaviour of algorithm changes with increase in value of n.
- This change in behaviour of the algorithm and algorithm's efficiency can be analysed by considering the the highest order of n.

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$$

| $n$ | $\sqrt{n}$ | $log_2 n$ | $n$ | $n\ log_2 n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 2 | 1 |
| 2 | 1.4 | 1 | 2 | 2 | 4 | 4 | 4 | 2 |
| 4 | 2 | 2 | 4 | 8 | 16 | 64 | 16 | 24 |
| 8 | 2.8 | 3 | 8 | $2.4 \cdot 10^1$ | 64 | $5.1 \cdot 10^2$ | $2.6 \cdot 10^2$ | $4.0 \cdot 10^4$ |
| 10 | 3.2 | 3.3 | 10 | $3.3 \cdot 10^1$ | $10^2$ | $10^3$ | $10^3$ | $3.6 \cdot 10^6$ |
| 16 | 4 | 4 | 16 | $6.4 \cdot 10^1$ | $2.6 \cdot 10^2$ | $4.1 \cdot 10^3$ | $6.5 \cdot 10^4$ | $2.1 \cdot 10^{13}$ |
| $10^2$ | 10 | 6.6 | $10^2$ | $6.6 \cdot 10^2$ | $10^4$ | $10^6$ | $1.3 \cdot 10^{30}$ | $9.3 \cdot 10^{157}$ |
| $10^3$ | 31 | 10 | $10^3$ | $1.0 \cdot 10^4$ | $10^6$ | $10^9$ | Very big computation | |
| $10^4$ | $10^2$ | 13 | $10^4$ | $1.3 \cdot 10^5$ | $10^8$ | $10^{12}$ | | |
| $10^5$ | $3.2 \cdot 10^2$ | 17 | $10^5$ | $1.7 \cdot 10^6$ | $10^{10}$ | $10^{15}$ | | |
| $10^6$ | $10^3$ | 20 | $10^6$ | $2.0 \cdot 10^7$ | $10^{12}$ | $10^{18}$ | | |

## Asymptotic Notations:

Asymptotic notations are the notation using which two algorithms can be compared with respect to efficiency based on the order of growth of an algorithms basic operation.

### Types of Notations:

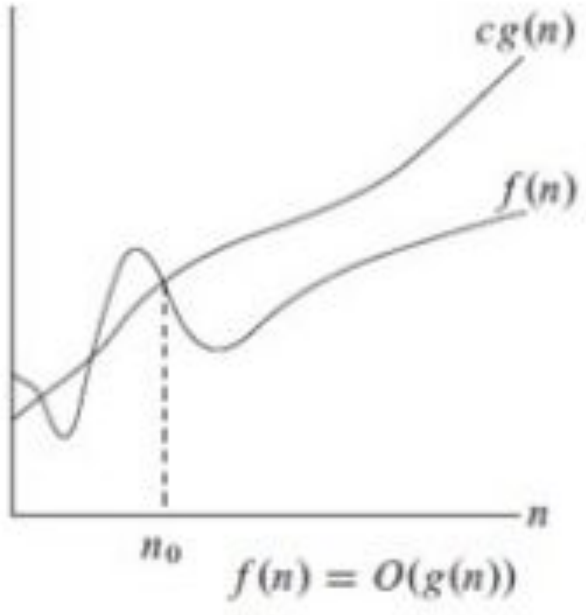
- Big Oh (O)
- Big Omega ( Ω)
- Big Theta (θ)

# Big - Oh Notation (O-Notation)

A function f(n) can be represented as the order of g(n) that is O(g(n)), if there exists a value of positive integer n as n0 and a positive constant c such that it should satisfy the following constraint :

$f(n) \leqslant c.g(n) \quad , \quad \forall\ n >= n0$ in all case

Hence, function g(n) is an upper bound for function f(n), as g(n) grows faster than f(n).



**Definition:** Let f(n) be the time efficiency of an algorithm. The function f(n) is said to be O(g(n)) [read as big-oh of g(n)], denoted by

$$f(n) \in O(g(n))$$

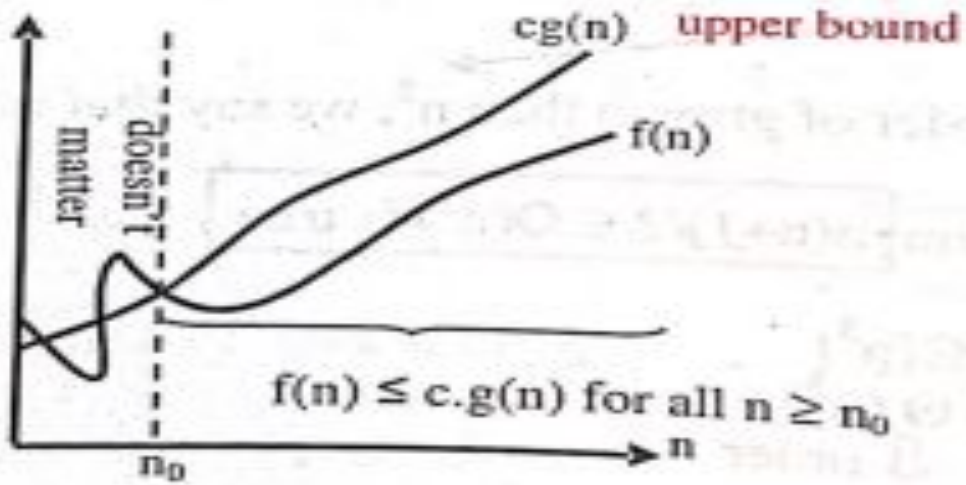
or

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

if and only if there exists a positive constant $c$ and positive integer $n_0$ satisfying the constraint

$$f(n) \leq c*g(n) \text{ for all } n \geq n_0 .$$

So, if we draw the graph f(n) and c*g(n) verses $n$, the graph of the function f(n) lies below the graph of c*g(n) for sufficiently large value of n as shown below:



# Example:

1. Let f(n) = 4n + 3 and g(n) =n ?

   Is f(n) = O(g(n)) ?

1. Let f(n) = 100n + 5. Express f(n) using Big - Oh.

2. Let f(n) = $10n^3$+8. Express f(n) using Big - Oh.

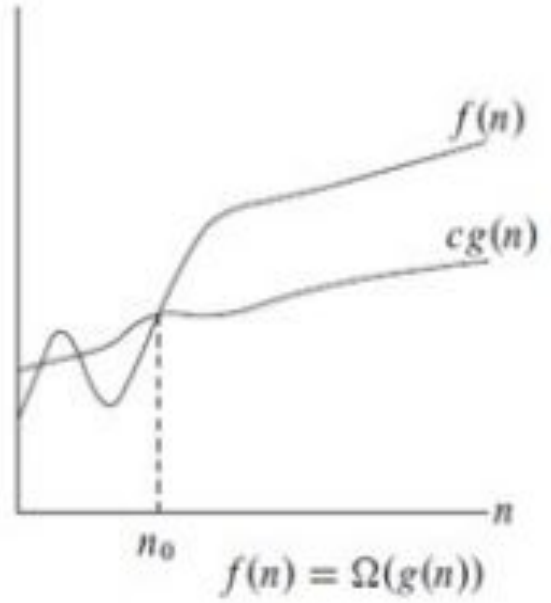
**Solutions :**
https://drive.google.com/file/d/1vjllO2hBburTP7ekpl8GxjDyHz0GPIEe/view?usp=sharing

# Big - Omega Notation (Ω-Notation)

A function f(n) can be represented as the order of g(n) that is Ω(g(n)), if there exists a value of positive integer n as n0 and a positive constant c such that it should satisfy the following constraint :

**f(n) ≥c.g(n) , ∀ n>=n0** in all case

Hence, function g(n) is an lower bound for function f(n), as f(n) grows faster than g(n).



**Definition:** Let f(n) be the time complexity of an algorithm. The function f(n) is said to be Ω(g(n)) [read as big-omega of g(n)] which is denoted by

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

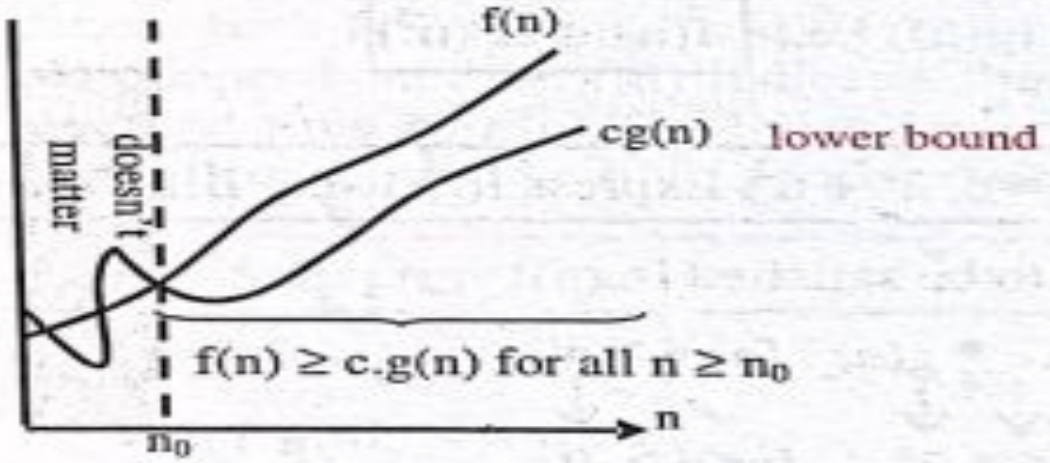
or

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

if and only if there exists a positive constant $c$ and non-negative integer $n_0$ satisfying the constraint

$$f(n) \geq c*g(n) \text{ for all } n \geq n_0 .$$

So, if we draw the graph f(n) and c*g(n) verses $n$, the graph of f(n) lies above the graph of g(n) for sufficiently large value of n as shown below:



# Example:

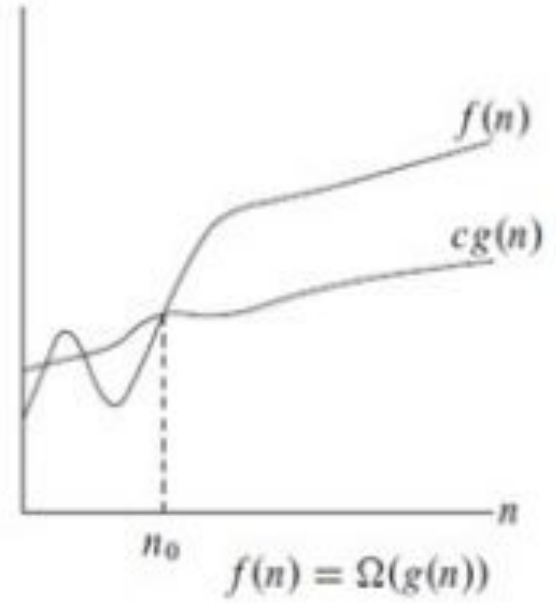
1. Let f(n) = 100n + 5. Express f(n) using Big - Omega

2. Let f(n) = $10n^3$+5. Express f(n) using Big - Omega

**Solutions :**
https://drive.google.com/file/d/1vjIlO2hBburTP7ekpl8GxjDyHz0GPIEe/view?usp=sharing

# Theta Notation (θ-Notation)

Here, f(n)=θ(g(n)) when there exist constants **c1** and **c2** that $C_1.g(n) \leqslant f(n) \leqslant C_2.g(n)$ for all sufficiently large value of **n**.

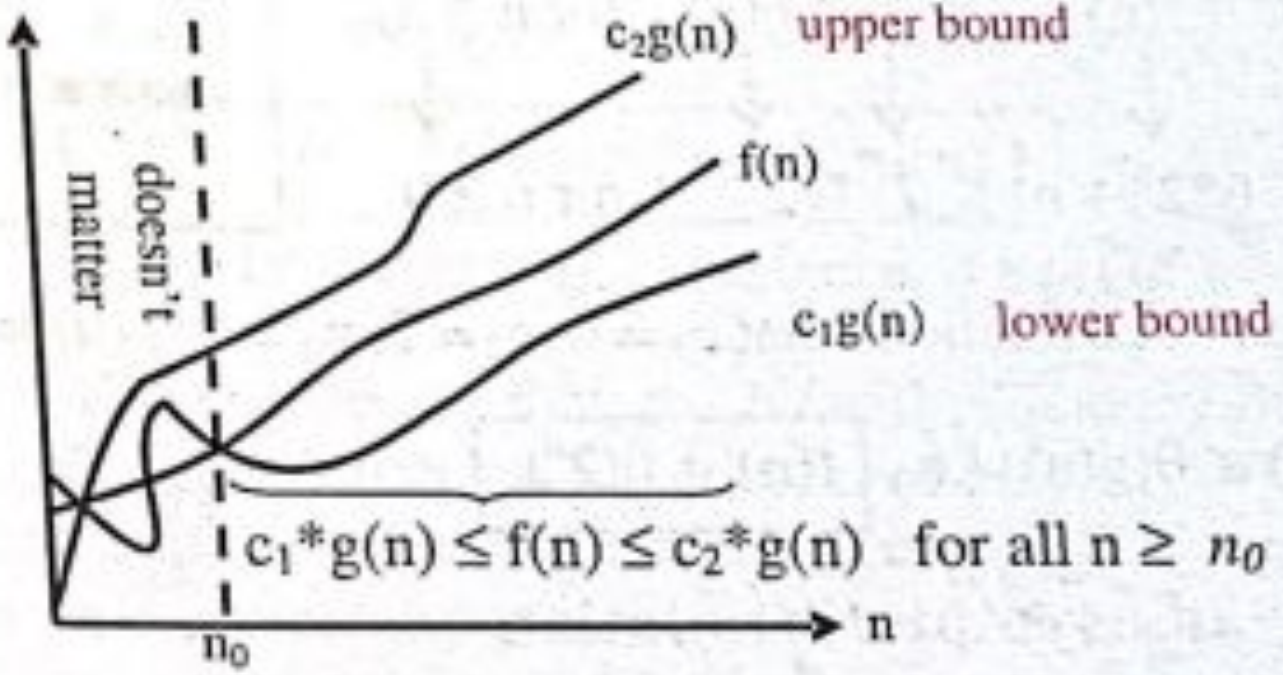



**C1.g(n) ⩽ f(n) ⩽ C2.g(n) ,∀ n>=n0** in all case

Here **n** is a positive integer. This means function **g** is

$c_1*g(n) \leq f(n) \leq c_2*g(n)$ for all $n \geq n_0$.

So, if we draw the graph f(n), $c_1*g(n)$ and $c_2*g(n)$ verses *n*, the graph of function f(n) lies above the graph of $c_1*g(n)$ and lies below the graph of $c_2*g(n)$ for sufficiently large value of n as shown below:



# Example :

1. Let f(n) = 100n + 5. Express f(n) using Big - Theta

2. Let f(n) = $10n^3$+5. Express f(n) using Big - Theta

**Solutions :**
https://drive.google.com/file/d/1vjIlO2hBburTP7ekpl8GxjDyHz0GPIEe/view?usp=sharing

# Mathematical Analysis of Non-Recursive Algorithms :

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation (in the innermost loop).
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its order of growth.

# Maximum of N Elements :

- To find the value of the largest element in a list of n numbers.

```
ALGORITHM :  MaxElement(A[ ],n)

//Determines the value of the largest element in a given array

//Input: An array A[0..n − 1] of real numbers

    //Output: The value of the largest element in A

    pos ←A[0]

for i ←1 to n − 1 do

      if (a[i] > a[pos])    pos←i

end for

  return pos
```

# Analysis of Algorithm :

- · The measure of an input's size here is the number of elements in the array, i.e., n.
- · There are two operations in the for loop's body: o The comparison A[i]> maxval and o The assignment maxval←A[i].
- · The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- · The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.

.

Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

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

i.e., Sum up 1 in repeated n-1 times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

# Element Uniqueness Problem :

To check whether all the elements in a given array are distinct.

```
ALGORITHM : UniqueElements (a [ ],n)
    for i ←0 to n − 2 do
        for j ←i + 1 to n − 1 do
            if A[i]= A[j ]
            return 0;
        end for
    end for
        return 1;
```

# Analysis of Algorithm :

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

# Other examples for Non Recursive Algorithm:

1. Matrix Multiplication
2. Linear Search

# Mathematical Analysis 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.

# Factorial of a Number

Compute the factorial function F(n) = n! for an arbitrary non-negative integer n.

Since n! = n*(n − 1)! , for n ≥ 1 and 0!= 1 by definition, we can compute

F(n) = F(n − 1) • n with the following recursive algorithm.

**n!=1 if n==0**

**n!=n*(n-1)! Otherwise**

$$F(n)=\begin{cases} 1 & \text{if } n==0 \\ n*f(n-1) & \text{Otherwise} \end{cases}$$

**Time Complexity : t(n) ∈ Θ(n)**

**ALGORITHM** Factorial(n)

//Computes n! Recursively

//Input: A nonnegative integer n

//Output: The value of n!

**if n = 0 return 1**

**else return n*fact(n − 1)**

# Other examples for Recursive Algorithm:

1. **Tower of Hanoi :** https://www.youtube.com/watch?v=q6RicK1FCUs
2. **Fibonacci Series :** https://www.youtube.com/watch?v=BNeOE1qMyRA

# Exercises:

1. What is an algorithm ? What are the criteria that all algorithm must satisfy?
2. What are the different ways of Computing GCD of two numbers?
3. Design an algorithm for Euclid's algorithmand also explain how algorithm halts eventually?
4. Find GCD(31415,14142) using Euclids/CIC/RS method.
5. What are the disadvantages of finding GCD using middle school procedure.
6. On what factors efficiency of algorithm depends on?
7. Explain Space and Time Complexity.
8. Define Basic Operations.

# Exercises:

1. What do you mean by asymptotic behaviour of a function ?
2. What are the different types of Asymptotic Function?
3. What is the general plan fo analysing non-recursive algorithm.
4. What is the general plan fo analysing recursive algorithm.
5. Design the algorithm for :
   a. Element Uniqueness Problem
   b. Matrix Multiplication
   c. Tower of Hanoi
   d. Maximum Element
   e. Fibonacci Series
   f. Linear Search
   g. Factorial of a number


Thank
you

JGI JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

# DEPARTMENT OF BCA

## ANALYSIS AND DESIGN OF ALGORITHMS

## (16BCA5D11)

## Module 1 : Role of Algorithms in Computing

**Prepared by: Prof. Bhavana Gowda D M**

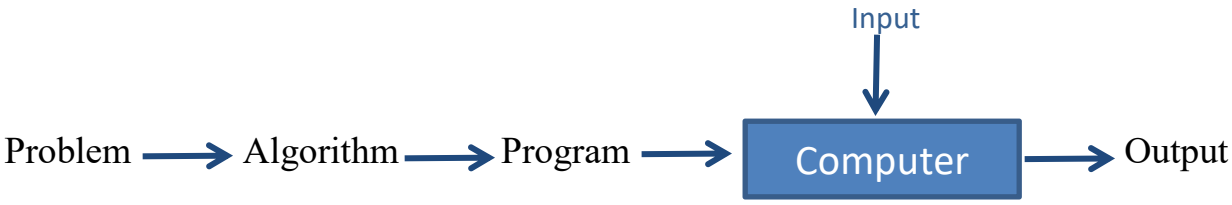
# UNIT 1: ROLE OF ALGORITHMS IN COMPUTING

## 1. Introduction

### What is an Algorithm?

An **algorithm** is defined as a sequence of finite unambiguous instructions to be followed to accomplish a given task in a finite number of steps by accepting a set of inputs and producing the desired output.

**Notions of an Algorithms :** Pictorial representation of an algorithm.




### Need for studying algorithms:

- The study of algorithms is the cornerstone of computer science.
- It can be recognized as the core of computer science.
- Computer programs would not exist without algorithms.
- With computers becoming an essential part of our professional & personal life‘s, studying algorithms becomes a necessity, more so for computer science engineers.
- Another reason for studying algorithms is that if we know a standard set of important algorithms, they further our analytical skills & help us in developing new algorithms for required applications.

## The Essential Characteristics of an Algorithm

The essential characteristics of an algorithm are;

1) Every step of an algorithm should perform a single task.
2) Ambiguity (Confusion) should not be included at any stage in an algorithm.
3) An algorithm should involve a finite number of steps to arrive at a solution.
4) Simple statement and structures should be used in the development of the algorithm.
5) Every algorithm should lead to a unique solution of the problem.
6) An algorithm should have the capability to handle some unexpected situations, which may arise during the solution of a problem (For example, division by zero).

## Properties of an Algorithm

All algorithms must satisfy the following criteria:

1) *Input*. Zero or more quantities are externally supplied.
2) *Output.* At least one Output is produced.
3) *Definiteness.* Each instruction should be simple and clear.
4) *Finiteness.* If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5) *Effectiveness.* Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper.

## Advantages of Algorithm

1. It is a step-by-step representation of a solution to a given problem, which is very easy to understand.
2. It has got a definite procedure, which can be executed within a set period of time.

3. It is easy to first develop an algorithm, and then convert it into a flowchart and then into a computer program.
4. It is independent of programming language.
5. It is easy to debug as every step has got its own logical sequence.

## Disadvantages of Algorithm

It is time consuming and cumbersome as an algorithm is developed first which is converted into a flowchart and then into a computer program.

## Example : Computing GCD (Greatest Common Divisor) :

- GCD of two numbers m and n denoted by GCD(m,n)
- Defined as the largest integer that divides both m and n such that the remainder is Zero and applicable only for positive integers.

Different Ways of Computing GCD of two nubers:

1. Euclid's Algorithm
2. Repetative Subtraction
3. Consecutive Integer Checking Algorithm
4. Middle School Procedure using prime factors

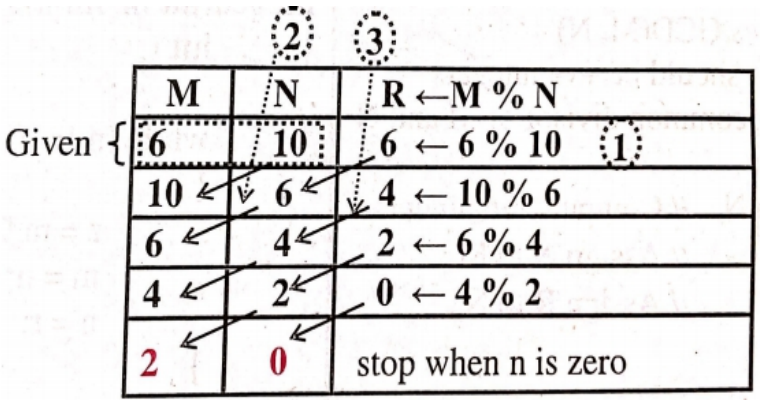
### 1. Euclid's Algorithm : Named after the mathematician - Euclid, Alexandria

**Procedure :**

Step1 : Compute the remainder using the statement R <- M % N

Step 2: Assign N to M i.e., M <- N

Step 3: Assign remainder R to N i.e., N <- R



| | M | N | R ← M % N |
|---|---|---|---|
| Given | 6 | 10 | 6 ← 6 % 10 (1) |
| | 10 | 6 | 4 ← 10 % 6 |
| | 6 | 4 | 2 ← 6 % 4 |
| | 4 | 2 | 0 ← 4 % 2 |
| | 2 | 0 | stop when n is zero |

**Iterative Algorithm :**

Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N

While N ≠ 0 do
    R ⟵ M%N
    M ⟵ N
    N ⟵ R
End While
Return M

**Recursive Algorithm :**

Algorithm GCD(M,N)

//Description : Computes GCD(M,N)

//Input : M and N should be positive integers

//Output: GCD of M and N

$$GCD(M,N) = \begin{cases} M & \text{if } N=0 \\ GCD(N, M\%N) & \text{Otherwise} \end{cases}$$

## Consecutive Integer Checking :

WKT GCD of 2 numbers M and N cannot be more than the smaller of these two numbers.

To start with : small = min(m,n)

Divide m and n by small.

if remainder in both the case is zero, then small will be the GCD.

Otherwise, decrement small by 1 and repeat the process till both m and n are divisible by small.

| small | m % small | n %small | small is GCD ? |
|---|---|---|---|
| 6 | 10 % 6 = 4 | 6 % 6 = 0 | 6 is not GCD |
| 5 | 10 % 5 = 0 | 6 % 5 = 1 | 5 is not GCD |
| 4 | 10 % 4 = 2 | 6 % 4 = 2 | 4 is not GCD |
| 3 | 10 % 3 = 1 | 6 % 3 = 0 | 3 is not GCD |
| 2 | 10 % 2 = 0 | 6 % 2 = 0 | 2 is GCD (remainder is 0) |

```
Algorithm GCD(M,N)
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N
Small        min(m,n)
while(1)
          if (m mod small=0)
            if (n mod small =0)
                  return small
            end if
          end if
small       small - 1
end while
return small
```

**Note : This algorithm will not work if one of the input is Zero.**

## 2. Repetative Subtraction :

Procedure :

- If m is greater than n, perform m-n and store in m

- If n is greater than m, perform n-m and store in n
- Repeat the above 2 steps as long as m and n are not equal
- if m and n are same, return m and n as GCD.

| M | N | Description |
|---|---|---|
| 10 | 6 | M = 10 – 6 = 4 (Since M > N, subtract N from M and store in M) |
| 4 | 6 | N = 6 – 4 = 2 (Since N > M, subtract M from N and store in N) |
| 4 | 2 | M = 4 – 2 = 2 (Since M >N, subtract N from M and store in M) |
| 2 | 2 | Since M and N are same, the GCD will be either M or N which is 2. |

GCD = 2.

**Algorithm GCD(M,N)**

```
//Description : Computes GCD(M,N)
//Input : M and N should be positive integers
//Output: GCD of M and N
while m ≠ n do
        if (m>n)
                m← m-n
          else
                n ← n-m
          end if
end while
return m
```

**Note :** If one of two numbers is a Zero. Return non-negative number as the GCD

## 3. Middle School Procedure:

**Procedure :**

Step 1: Find the prime factors of m

Step 2: Find the prime factors of n

Step 3: Identify the common prime factors obtained in step 1 and step 2

Step 4: Find the product of all common factors and return the result as gcd of two given numbers.

**Disadvantages of Middle School Procedure:**

- More Complex
- This algorithm does not specify how to generate the prime factors and hence it is not a legitimate algorithm.
- This algorithm does not specify how to find common factors and hence it is not possible to write the program.

## Fundamentals of Algorithmic Problem Solving

A sequence of steps involved in designing and analyzing an algorithm is shown in the figure below.

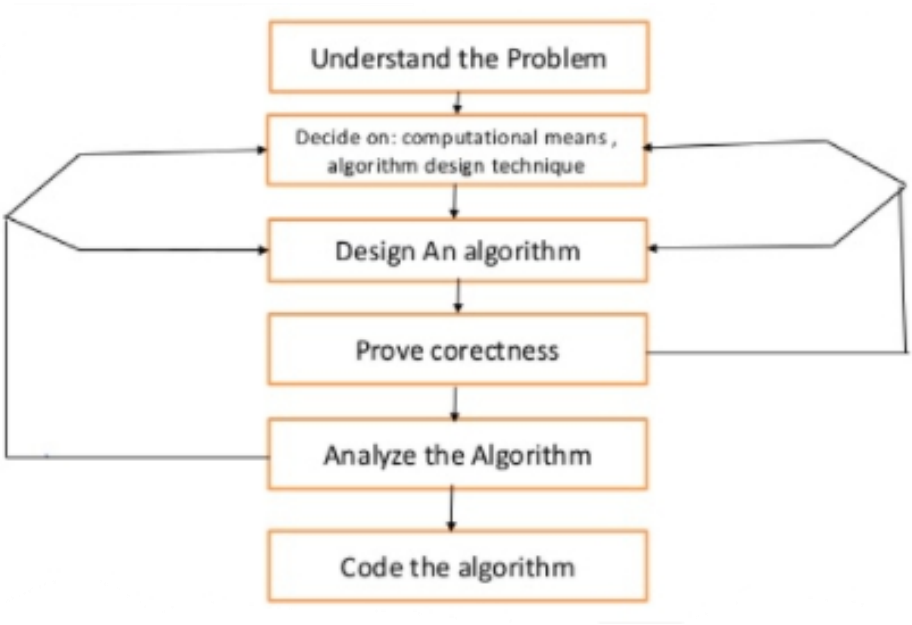



Figure: Flow diagram of designing and analyzing an algorithm

### *(i) Understanding the Problem*

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (instance) to the problem and range of the input get fixed.

### *(ii) Decision making*

The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device

- In random-access machine (RAM), instructions are executed one after another (The central assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.
- In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.
- Choice of computational devices like Processor and memory is mainly based on space and time efficiency

### (b) Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly and solving it approximately.
- An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
- If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. i.e., produces an approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

### (c) Algorithm Design Techniques

- An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

- Algorithms+ Data Structures = Programs
- Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- Implementation of algorithm is possible only with the help of Algorithms and Data Structures
- Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on

### *(iii) Methods of Specifying an Algorithm*

There are three ways to specify an algorithm. They are:

a. Natural language

b. Pseudo code

c. Flowchart

### *(iv) Proving an Algorithm's Correctness*

a. Once an algorithm has been specified then its correctness must be proved.

b. An algorithm must yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).

c. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

d. The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The error produced by the algorithm should not exceed a predefined limit

### *(v) Analyzing an Algorithm*

a. For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are:
b. Time efficiency, indicating how fast the algorithm runs, and
c. Space efficiency, indicating how much extra memory it uses.
d. The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
e. So factors to analyze an algorithm are:

- ♣ Time efficiency of an algorithm
- ♣ Space efficiency of an algorithm
- ♣ Simplicity of an algorithm
- ♣ Generality of an algorithm

### *(vi) Coding an Algorithm*

a. The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.
b. The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduce by inefficient implementation.
c. Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common sub-expressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
d. Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in

running time by orders of magnitude.

e. It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

## Role of algorithms in computing

The most important problem types related to computing are:

1) *Sorting*: The sorting problem is to rearrange the items of a given list in non-decreasing (ascending) order.

2) *Searching:* The searching problem deals with finding a given value, called a search key, in a given set.

3) String processing: A string is a sequence of characters from an alphabet. Strings comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four character alphabet {A, C, G, T}. It is very useful in bioinformatics. Searching for a given word in a text is called string matching.

4) *Graph problems*: A graph is a collection of points called vertices, some of which are connected by line segments called edges. Some of the graph problems are graph traversal, shortest path algorithm, topological sort, traveling salesman problem and the graph-coloring problem and so on.

5) *Combinatorial problems*: These are problems that ask, explicitly or implicitly, to find a combinatorial object such as a permutation, a combination, or a subset that satisfies certain constraints.

6) *Geometric problems:* Geometric algorithms deal with geometric objects such as points, lines, and polygons. Geometric algorithms are used in

computer graphics, robotics, and tomography.

7) Numerical problems are problems that involve mathematical equations, systems of equations, computing definite integrals, evaluating functions, and so on. The majority of such mathematical problems can be solved only approximately.

## Algorithms as a Technology.

There a huge number of real life applications where, algorithms play the major and vital role. Few of such applications are discussed below.

- The ***Internet*** without which it is difficult to imagine a day is the result of *clever and efficient algorithms*. With the aid of these algorithms, various sites on the Internet are able to manage and manipulate this large volume of data. Finding good routes on which the data will travel and using search engine to find pages on which particular information is present.

- Another great milestone is the Human Genome Project which has great progress towards the goal of identification of the 100000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up the human DNA, storing this *huge amount of information* in databases, and developing tools for data analysis. Each of these steps required *sophisticated and efficient algorithms.*

- The day-to-day ***electronic commerce*** activities is hugely dependent on our personal information such as credit/debit card numbers, passwords, bank statements, OTPs and so on. The core technologies used include public-key crypto currency and digital signatures which are based on numerical

algorithms and number theory.

- The approach of **linear programming** is also one such technique which is widely used like
  - In *manufacturing* and other commercial enterprises where resources need to be allocated scarcely in the most beneficial way.
  - Or a institution may want to determine where to spend money buying advertising in order to *maximize* the chances of their institution to grow.

- **Shortest path algorithm** also has an extensive use as
  - In a *transportation* firm such as a trucking or railroad company, may have financial interest in finding shortest path through a road or rail network because taking shortest path result in lower labour or fuel costs.
  - Or a *routing node* on the Internet may need to find the shortest path through the network in order to route a message quickly.

- Even an application that does not require algorithm content at the application level relies heavily on algorithms as the application depends on *hardware, GUI, networking or object orientation* and all of these make an extensive use of algorithms.

## Getting Started

## Fundamentals of the Analysis of Algorithm Efficiency

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

a. Analysis Framework.

b. Asymptotic Notations and its properties.

c. Mathematical analysis for Recursive algorithms.

d. Mathematical analysis for Non-recursive algorithms.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

### (a) Time Complexity

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

**Components that affect Time Efficiency :**

A. Speed of the Computer

B. Choice of the Programming Language

C. Compiler Used

D. Choice of the Algorithm

E. Size of Inputs / Outputs

### (b) Space Complexity

It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

**Components that Affects Space Efficiency :**

A. Program Space

B. Data Space

C. Stack Space

## The Need for Analysis

In this section, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

The algorithm analysis framework consists of the following:

- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

### *Measuring an Input's Size:*

An algorithm's efficiency is defined as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching.

## ***Units for Measuring Running Time:***

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code.
- The difficulty of clocking the actual running time of the program.

So, we need metric to measure an algorithm's efficiency that does not depend on these extraneous factors. One possible approach is **to count the number of times each of the algorithm's operations is executed**. This approach is excessively difficult.

## ***Orders of Growth***

A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. Some common orders of growth seen often in complexity analysis are:

| | |
|---|---|
| $O(1)$ | constant |
| $O(\log n)$ | logarithmic |
| $O(n)$ | linear |
| $O(n \log n)$ | "n log n" |
| $O(n^2)$ | quadratic |
| $O(n^3)$ | cubic |
| $n^{O(1)}$ | polynomial |
| $2^{O(n)}$ | exponential |

Table: Values (approximate) of several functions important for analysis of algorithms

| $n$ | $\sqrt{n}$ | $\log_2 n$ | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 2 | 1 |
| 2 | 1.4 | 1 | 2 | 2 | 4 | 4 | 4 | 2 |
| 4 | 2 | 2 | 4 | 8 | 16 | 64 | 16 | 24 |
| 8 | 2.8 | 3 | 8 | $2.4 \cdot 10^1$ | 64 | $5.1 \cdot 10^2$ | $2.6 \cdot 10^2$ | $4.0 \cdot 10^4$ |
| 10 | 3.2 | 3.3 | 10 | $3.3 \cdot 10^1$ | $10^2$ | $10^3$ | $10^3$ | $3.6 \cdot 10^6$ |
| 16 | 4 | 4 | 16 | $6.4 \cdot 10^1$ | $2.6 \cdot 10^2$ | $4.1 \cdot 10^3$ | $6.5 \cdot 10^4$ | $2.1 \cdot 10^{13}$ |
| $10^2$ | 10 | 6.6 | $10^2$ | $6.6 \cdot 10^2$ | $10^4$ | $10^6$ | $1.3 \cdot 10^{30}$ | $9.3 \cdot 10^{157}$ |
| $10^3$ | 31 | 10 | $10^3$ | $1.0 \cdot 10^4$ | $10^6$ | $10^9$ | Very big computation | |
| $10^4$ | $10^2$ | 13 | $10^4$ | $1.3 \cdot 10^5$ | $10^8$ | $10^{12}$ | | |
| $10^5$ | $3.2 \cdot 10^2$ | 17 | $10^5$ | $1.7 \cdot 10^6$ | $10^{10}$ | $10^{15}$ | | |
| $10^6$ | $10^3$ | 20 | $10^6$ | $2.0 \cdot 10^7$ | $10^{12}$ | $10^{18}$ | | |

### *Worst-Case, Best-Case, and Average-Case Efficiencies*

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.

**EXAMPLE**:

Consider the Sequential Search algorithm

SequentialSearch(A[0..n - 1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n - 1] and a search key K

//Output: The index of the first element in A that matches K or -1 if there

is no match i ←0

```
while i < n and A[i] ≠
K do i ←i + 1
if  i  <  n
return  i  else
return -1
```

Clearly, the running time of this algorithm can be quite different for the same list size n. In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

The **worst-case efficiency** of an algorithm is its efficiency for the worst case input of size n. The algorithm runs the longest among all possible inputs of that size.

The **best-case efficiency** of an algorithm is its efficiency for the best case input of size n. The algorithm runs the fastest among all possible inputs of that size n. In sequential search, If we search a first element in list of size n. (i.e. first element equal to a search key).

The **Average case efficiency** lies between best case and worst case. To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.

# Asymptotic notation and Basic Efficiency Classes

## Asymptotic Notation:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Let t(n) and g(n) can be any nonnegative functions defined on the set of natural numbers. The algorithm's running time t(n) usually indicated by its basic operation count C(n), and g(n), some simple function to compare with the count.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta

### Big Oh (O): Asymptotic Upper Bound

'O' (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that −

$$f(n) \leqslant c.g(n) \quad \text{for} \quad n > n_o \quad \text{in all case}$$

Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

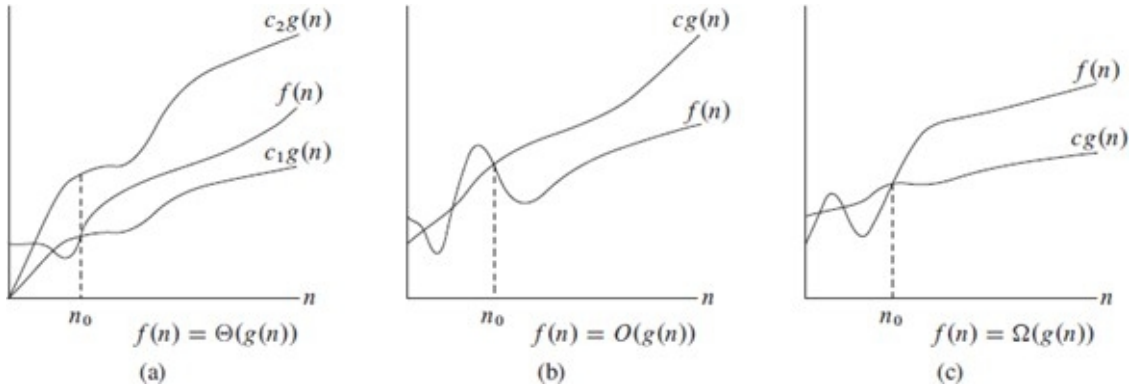
**Example**

Let us consider a given function, $f(n)=4.n^3+10.\ 4.n^2+5.n+1$

Considering g(n)=n3,

$$f(n) \leqslant 5.g(n) \text{ for all the values of } n>2$$

Hence, the complexity of ***f(n)*** can be represented as O(g(n)), i.e. $O(4.n^3)$




**Omega (Ω): Asymptotic Lower Bound**

We say that f(n)=Ω(g(n)) when there exists constant **c** that f(n)⩾ c.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

**Example**

Let us consider a given function, f(n)= $4.n^3+10.\ 4.n^2+5.n+1$

Considering g(n)= $^3$ , f(n)⩾ 4.g(n) for all the values of n>0

Hence, the complexity of ***f(n)*** can be represented as Ω(g(n)), i.e. Ω( $^3$)

**Theta (θ): Asymptotic Tight Bound**

We say that f(n)=θ(g(n)) when there exist constants $\mathbf{c_1}$ and $\mathbf{c_2}$ that $_1$.g(n)⩽ f(n)⩽ $_2$.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer. This means function **g** is a tight bound for function **f**.

**Example**

Let us consider a given function, f(n)= $4.n^3+10.\ 4.n^2+5.n+1$

Considering g(n)= $^3$, 4.g(n)⩽ f(n)⩽ 5.g(n) for all the large values of **n**.

Hence, the complexity of ***f(n)*** can be represented as θ(g(n)), i.e. θ($n^3$).

**Asymptotic Analysis**

*Solving Recurrence Equations*

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-conquer paradigm.

A recurrence relation can be solved using the following methods −

- **Substitution Method** − In this method, we guess a bound and using mathematical induction we prove that our assumption was correct.
- **Recursion Tree Method** − In this method, a recurrence tree is formed where each node represents the cost.
- **Master's Theorem** − This is another important technique to find the complexity of a recurrence relation.

***General Plan for Analyzing 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.

***EXAMPLE 1***: Compute the factorial function F(n) = n! for an arbitrary non-negative integer n. Since n! = 1•. . . . • (n − 1) • n = (n − 1)! • n, for n ≥ 1 and 0!= 1 by definition, we can compute F(n) = F(n − 1) • n with the following recursive algorithm.

**ALGORITHM** Factorial(n)
//Computes n! Recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return Factorial(n − 1) * n

## Algorithm analysis

- For simplicity, we consider n itself as an indicator of this algorithm's input size. i.e. 1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote M(n). Since the function F(n) is computed according to the formula F(n) = F(n −1)*n for n > 0.
- The number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\text{To compute F(n-1)}} + \underbrace{1}_{\text{To multiply F(n-1) by n}} \quad \text{for } n > 0$$

  - M(n − 1) multiplications are spent to compute F(n − 1), and one more multiplication is needed to multiply the result by n.

## Recurrence relations

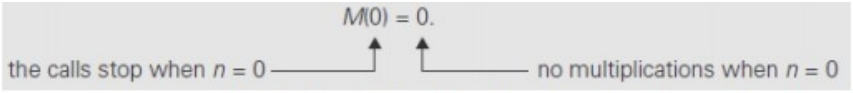
The last equation defines the sequence M(n) that we need to find. This equation

defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called recurrence relations or recurrences.

Solve the recurrence relation M(n)=M(n-1)+1 , i.e., to find an explicit formula for M(n) in terms of n only. To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if n = 0 return 1.

This tells us two things. First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0. Second, by inspecting the pseudo code's exiting line, we can see that when n = 0, the algorithm performs no multiplications.




Thus, the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$M(n) = M(n-1) + 1$ for $n > 0$,
$M(0) = 0$ for $n = 0$.

**Method of backward substitutions**

$M(n) = M(n-1) + 1$ substitute $M(n-1) = M(n-2) + 1$
$= [M(n-2) + 1] + 1$
$= M(n-2) + 2$ substitute $M(n-2) = M(n-3) + 1$
$= [M(n-3) + 1] + 2$
$= M(n-3) + 3$
...
$= M(n-i) + i$
...
$= M(n-n) + n$
$= n$.

Therefore $M(n)=n$

***General Plan for Analyzing the Time Efficiency of Non-recursive Algorithms***

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation (in the innermost loop).
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its order of growth.

**EXAMPLE 1**: Consider the problem of finding the value of the largest element in a list of n numbers. Assume that the list is implemented as an array for simplicity.

```
ALGORITHM MaxElement(A[0..n − 1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in
A maxval ←A[0]
for i ←1 to n − 1 do
        if A[i]>maxval
        maxval←A[i]
End for
   return maxval
```

**Algorithm Analysis**

- The measure of an input's size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop's body: o The comparison A[i]> maxval and o The assignment maxval←A[i].
- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n

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

i.e., Sum up 1 in repeated n-1 times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

− 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

**EXAMPLE 2:**

Consider the element uniqueness problem: check whether all the Elements in a given array of n elements are distinct.

```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
 //Input: An array A[0..n − 1]
//Output: Returns “true” if all the elements in A are distinct and
“false” otherwise
for i ←0 to n − 2 do
for j ←i + 1 to n − 1 do
        if A[i]= A[j ]
         return 0;
end for
end for
```

**Algorithm Analysis:**

- The measure of an input’s size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop’s body: o The comparison A[i]> maxval and o The assignment maxval←A[i].
- The comparison operation is considered as the algorithm’s basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop’s variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

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

i.e., Sum up 1 in repeated n-1 times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

**Algorithm Analysis:**

The natural measure of the input's size here is again n (the number of elements in the array).

Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.

The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n −2

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

**Algorithm Design :**

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

## Problem Development Steps

The following steps are involved in solving computational problems.

Problem definition

Development of a model

Specification of an Algorithm

Designing an Algorithm

Checking the correctness of an Algorithm

Analysis of an Algorithm

Implementation of an Algorithm

Program testing

Documentation

JAIN DEEMED-TO-BE UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND IT

# DEPARTMENT OF BCA

# ANALYSIS AND DESIGN OF ALGORITHMS (16BCA5D11)

## Module 2 : Brute Force Approaches

**Prepared by: Prof. Bhavana Gowda D M**

# UNIT 2: BRUTE FORCE APPROACHES

## Introduction

### What is Brute Force Method?

The straightforward method of solving a given problem based on the problems statement and definition is called Brute Force method.

Ex : Selection Sort, Bubble Sort, Linear Search etc.

### Pros and Cons of Brute Force Method :

**Pros:**

- The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.
- It is a generic method and not limited to any specific domain of problems.
- The brute force method is ideal for solving small and simpler problems.
- It is known for its simplicity and can serve as a comparison benchmark.

**Cons:**

- The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the O(N!) order of growth.
- This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.
- Brute force algorithms are slow.
- Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.

### Exhaustive Search :

A Search problem involves some set of possibilities and we are looking for one or more of the possibilities and we are looking for one or more of the possibilities that satisfy some property.

Example : Travelling Salesperson Problem

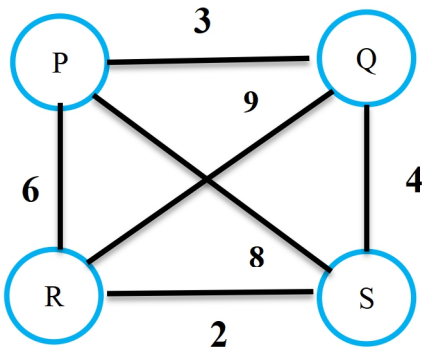
**Method :**

1. Generate a list of all potential solutions to the problem.
2. Evaluate potential solutions one by one disqualifying infeasible ones and for an optimization problem,keeping track of the best one found so far.

## Travelling Salesman Problem :

- ✓ The Travelling Salesman Problem (TSP) is an optimization problem used to find the shortest path to travel through the given number of cities.
- ✓ Travelling salesman problem states that given a number of cities N and the distance between the cities, the traveler has to travel through all the given cities exactly once and return to the same city from where he started and also the length of the path is minimized.
- ✓ The Travelling Salesman Problem (TSP) can be formulated as follows: to choose a pathway optimal by the given criterion. In this, optimal criterion is usually the minimal distance between towns or minimal travel expenses. Travelling salesman should visit a certain number of towns and return to the place of departure, so that they visit each town only once.
- ✓ TSP can be solved as shown below:
    1. Get all the routes from one city to another city by taking various permutations.
    2. Compute the route length for each permutation and select the shortest among them.
- ✓ TSP can be modeled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (i.e. each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

**Example :**




Here P Q R S are Cities.

- ✓ Distance between various cities are represented as numbers in each of the edge.
- ✓ Let assume, Salesperson starts from City P then the various routes using which he can visit each and every city exactly once and returns back to the start city P along with the cost is as shown below:

**P ⟶ Q ⟶ R ⟶ S ⟶ P (COST = 22)**
**P ⟶ Q ⟶ S ⟶ R ⟶ P (COST = 15)**
**P ⟶ R ⟶ Q ⟶ S ⟶ P (COST = 27)**
**P ⟶ R ⟶ S ⟶ Q ⟶ P (COST = 15)**
**P ⟶ S ⟶ Q ⟶ R ⟶ P (COST = 27)**
**P ⟶ S ⟶ R ⟶ Q ⟶ P (COST = 22)**

Finally considering routes with minimum cost to get the maximum profit, we will consider the below routes and cost:

**P ⟶ Q ⟶ S ⟶ R ⟶ P (COST = 15)**
**P ⟶ R ⟶ S ⟶ Q ⟶ P (COST = 15)**

**Note :**

**In general, for n cities number of routes=(n-1)! I.e., f(n)=f(n-1)!**

**Hence the time complexity is given by ,**

**F(n)=O(n!)**

# Selection Sort and Bubble Sort

## Selection Sort

- This is a sorting algorithm, which is very simple to understand and implement.
- The algorithm achieves its name from the fact that with each iteration the appropriate value for a key position is selected from the list of remaining elements and put in the required position of the array.
- However the algorithm is not efficient for large arrays.
- The method of selection sort relies heavily on a comparison mechanism to achieve its goals.

**Procedure :**

1. As the name indicates, we first find the smallest eement in the list and we exchange it with the first item.

2. Obtain the second smallest element in the list and exchange it with the second element and so on.

3. Finally all the items will be arranged in ascending order.

4. This technique is called Selection Sort.

```
Algorithm : selectionsort(a[ ], n)
//purpose : sort the given elements in ascending orderusing
selection sort.
//inputs : n - number of elements
           a - the items to be sorted are present in the array.
//outputs : a - contains the sorted list

for  i=0 to n-2 do
        pos = i;
        for j=i+1 to n-1 do
                if (a[j]<a[pos])
                pos <-- j;
        end for
        temp=a[pos]

        a[pos]=a[i]

        a[i]=temp

end for
```

## Analysis of Selection Sort : $\theta (n^2)$

## BUBBLE SORT (Sinking Sort)

- The algorithm of bubble sort functions as follows.
- The algorithm begins by comparing the element at the bottom of the array with the next element.
- If the first element is larger than the second element then they are swapped or interchanged.
- The process is then repeated for the next two elements. After n-1 comparisons the largest of all the items slowly ascends to the top of the array.
- The entire process till now forms one pass of comparisons.
- During the next pass the same steps are repeated from the beginning of the array, however this time the comparisons are only for n-1 elements.
- The second pass results in the second largest element ascending to its position. The process is repeated again and again until only two elements are left for comparison.
- The last iteration ensures that the first two elements of the array are placed in the correct order.

**Algorithm :**

**Algorithm : Bubblesort (a[ ],n)**

```
//Purpose : Arrange numbers in Ascending Order
//Inputs : n - number of items present in the table
        A - the items to be sorted are present in the table
//Output: a - contains the sorted list.
for j=1 to n-1 do
        for i=0 to n-j-1 do
                if(a[i] > a[j+1])
                        temp = a[i]
                        a[i] = a[i+1]
                        a[i+1] = a[temp]
                end if
        end for
end for
```

**Time Complexity of BubbleSort : $O(n^2)$**

## Sequential Search (Linear Search) :

- The simplest of all forms of searching is the linear search.
- This search is applicable to a table organized either as an array or as a linked list.
- Let us assume that A is an array of N elements from A[0] through A[N - 1].
- Let us also assume that ele is the search element. The search starts by sequentially comparing the elements of the array one after the other from the beginning to the end with the element to be searched.
- If the element is found its position is identified otherwise an appropriate message is displayed.

## Algorithm :

**Algorithm : Linear Search (key,a[ ],n)**

//Purpose : This algorithm searches for key element

//Inputs : n - number of elements present in the array

A - elements in the array where searching takes place

Key - item to be searched

//Outputs : The function returns the position of the key if found

Otherwise, function returns -1 indicating search is unsuccessful.

```
for i=0 to n-1 do
    if( key = a[i] )
    return i;
    end if
end for
return -1;
```

**Time Complexity : O(N)**

# Sorting, Sets and Selection:

## Sorting :

The process of rearranging the given elements in ascending order or descending order is called sorting.

## Divide and Conquer Method :

- It is a top-down approach for designing algorithms which consist of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find.
- The solutions of all smaller problems are then combined to get a solution for the original problem.

Divide and Conquer method involves 3 steps:

A. **Divide :** Problem is divided into a number of sub problems.

B. **Conquer :** If the sub problem are smaller in size, the problem can be solved using straightforward method. If the sub problem size is large, then they are divided into number of sub problems of the same type and size. Each sub problem is solved recursively.

C. **Combine :** The solution od sub problems are combined to get the solution for the larger problem.

## 1. Merge Sort :

- This sorting method follows the technique of **divide-and-conquer.**
- The technique works on a principle where a given set of inputs are split into distinct subset and the required method is applied on each subset separately i.e., the sub problems are first solved and then the solutions are combined into a solution of the whole.
- Most of the times the sub problems generated will be of the same type as the original problem.
- In such situations re-application of the divide-and-conquer technique may be necessary on each sub problem.

- This is normally achieved through a recursive procedure.
- Thus smaller and smaller sub problems are generated until it is possible for us to solve each sub problem without splitting.

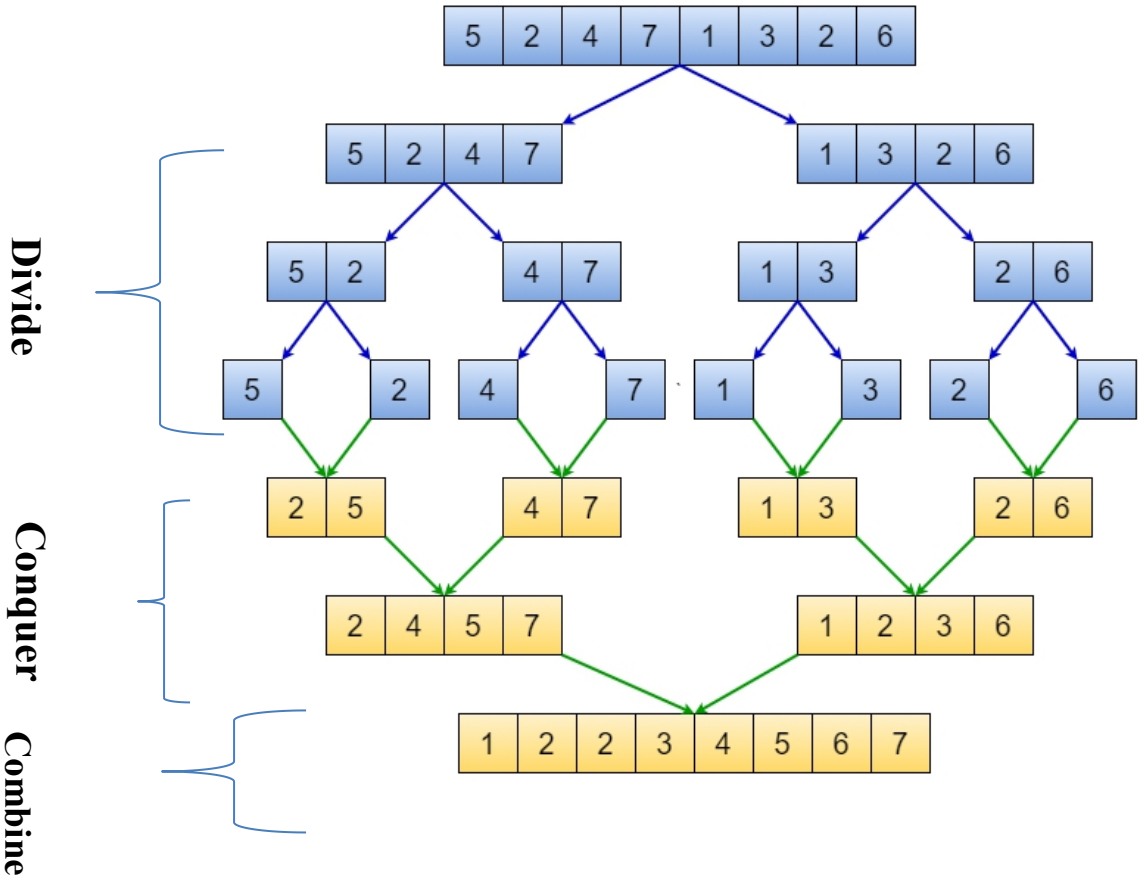
## The Steps involved in Merge Sort :

1. **Divide :** Divide the given array consisting of n elements into two parts of n/2 elements each.
2. **Conquer :** Sort the left part of the array and right part of the array recursively using mergesort.
3. **Combine/Merge :** Merge the sorted left part and sorted right part to get a single sorted array.

**Example :**

**N = 8**

**Elements : 5, 2, 4, 7, 1, 3, 2, 6**



## Algorithm :

**Algorithm MergeSort(a,low,high)**

//Purpose :Sort the given array between lower bound and upper bound

//Inputs: a is an array consisting of unsorted elements with low and high as lower bound and upper bound

//Output a:It is an array consisting of sorted elements

```
if (low>high) return //No elements to partition
mid<- (low+high)/2 //Divide the array into two parts
MergeSort (a,low,mid) //Sort the left part of the array
MergeSort (a,mid+1,high)//Sort the right part of the array
SimpleMerge (a,low,mid,high) // Merge the left part and right part
//End of the algorithm MergeSort
```

**Simple Merge (a,low,mid,high)**

//Purpose: Merge two sorted arrays where the first array starts from low to mid and the second

// starts from mid+1 to high

//Input : a is sorted from the index position low to mid

// a is sorted from index position mid+1 to high

//Output : a is sorted from index low to high

```
i<-low
J<- mid+1
k<-low
```

```
while(i<=mid and j<=high)
        if( a[i] < a[j] ) then
                c[k] <- a[i]
                i<-i+1; k<-k+1 ;
        else
                c[k]<-a[j]
                j<-j+1 ; k<-k+1 ;
        end if
end while
while(i<=mid)
        c[k]<-a[i]
        k <- k+1, i <- i+1
end while
while(j<=high)
        c[k]<-a[j]
        K <- k+1, j <- j+1;
end while
for i=low to high // Copy the elements from c to a a[i]<- c[i]
end for     //End of Algorithm Simple Merge
```

**Analysis of Merge Sort :**

$T(n) = \theta ( n\log_2 n)$

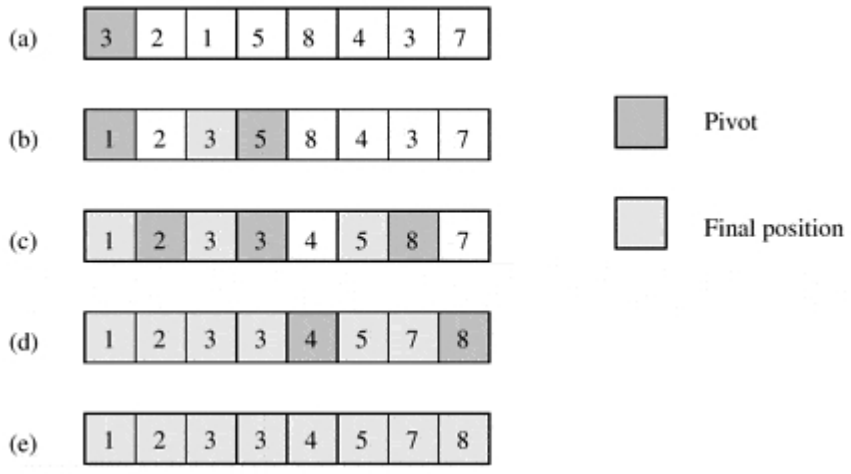
## 2. Quick Sort

This is one of the best techniques for a large set of data. This technique also works on the method of partitioning. The process of divide and conquer is again recursively applied. In the merge sort method of sorting, the array was partitioned exactly at the midpoint, however in this method of sorting the partition is created at a position such that the elements to the left of the partition is less than the elements to the right of the partition. The entire process functions as follows. First we choose an element from a specific position in the array to be sorted (for example let the element "ele" be chosen as the first element of the array "a" so that ele = a[0]). Suppose that the elements of the array a are partitioned so that ele is placed into the position "p" and the following conditions are satisfied.

- Each of the elements in the positions 0 through p-1 is less than or equal to ele.
- Each of the elements in the positions j+1 through n-1 is greater than or equal to ele.

The purpose of quick sort is to move the data item in the correct direction jus enough for it to reach its final place. Thus the amount of swapping is greatly reduced and the required item moves a greater distance in a shorter duration of time.

**Example :**



## Algorithm :

**Algorithm : QuickSort(a,low,high)**

//Purpose :Sort the given array using quicksort

//Inputs: low: The position of first element in array a

high: The position of the last element of array a

a: It is an array consisting of unsorted elements

//Output a:It is an array consisting of sorted elements

**if (low>high) return //No elements to partition**

**k<--partition(a,low,high)** //Divide the array into two parts

**QuickSort(a,low,k-1) //Sort the left part of the array**

**QuickSort(a,k+1,high) //Sort the right part of the array**

"**algorithm QuickSort Ends here**"

**Algorithm : partition(a,low,high)**

key<-a[low]

i<-low

j<-high+1

**while**(i<=j)

**do** i<-i+1 **while**(key>=a[i])

**do** j<-j-1 **while**(key<a[j])

**end while**

If (i<j) exchange(a[low],a[j])

**return** j **//End of the algorithm Partition**

## Analysis of Quick Sort :

**Best Case :** $T(n) = \theta ( n\log_2 n)$

**Worst Case :** $T(n) = O ( n^2 )$

**Average Case :** $T(n) = O ( n\log_2 n )$

## 3. Bucket sort

- **Bucket sort**, or **bin sort**, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.
- Several buckets are created. Each bucket is filled with a specific range of elements. The elements inside the bucket are sorted using any other algorithm. Finally, the elements of the bucket are gathered to get the sorted array.
- The process of bucket sort can be understood as a scatter-gather approach. The elements are first scattered into buckets then the elements of buckets are sorted. Finally, the elements are gathered in order.

**Bucket sort works as follows:**

- Set up an array of initially empty "buckets".
- Scatter: Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather: Visit the buckets in order and put all elements back into the original array.

```
BucketSort ( )
Create N buckets each of which can hold a range of values
    for all the buckets
      initialize each bucket with 0 values
      for all the buckets
      put elements into buckets matching the range
      for all the buckets
      sort elements in each bucket
      gather elements from each bucket
End bucket sort
```

## Example :


**Time Complexity :**

**Worst Case Complexity: O($n^2$)**

**Best Case Complexity: O(n+k)**

**Average Case Complexity: O(n)**

# Radix Sort :

- Radix sort is a small method that many people intuitively use when alphabetizing a large list of names.
- Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.
- Intuitively, one might want to sort numbers on their most significant digit.
- However, Radix sort works counter-intuitively by sorting on the least significant digits first.
- On the first pass, all the numbers are sorted on the least significant digit and combined in an array.
- Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

**Example :**

## Time Complexity :

n: number of elements

k: the range of the keys for each number. We will also repeat the operation for this amount.

All of the Time Complexities of Radix Sort is always **O(n*k)**

## Algorithm :

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort
countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```