Module - 1

Syllabus :-

Introduction : What is an algorithm? Fundamentals of algorithmic problem solving.

Fundamentals of the analysis of algorithm efficiency : Analysis framework, Asymptotic notations and Basic efficiency classes, Mathematical analysis of non recursive algorithms, Mathematical analysis of recursive algorithms.

Brute Force Approaches : Selection sort and Bubble sort, sequential search and brute force string matching.

What is an algorithm?

- It is defined as unambiguous, step by step procedure to solve a given problem in finite number of steps by accepting a set of inputs and producing the desired output.
- After producing the result, the algorithm should terminate
- Definition of an algorithm is pictorially represented as

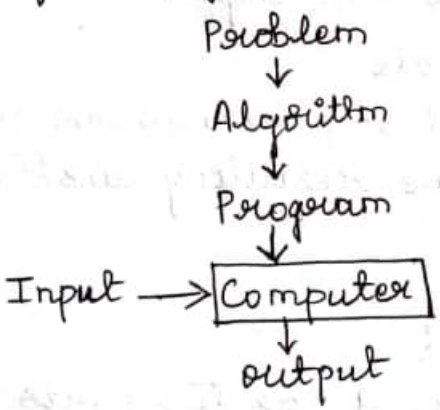


Fig :- Notion of algorithm

- Solution to a given problem is expressed in the form of an algorithm.
- Algorithm is converted into program.
- Program when it is executed, accept the input and produce the desired output.

Properties of an algorithm :-

An algorithm must satisfy the following criteria

1) Unambiguous : Algorithm should be clear. Each of its steps should be clear and must lead to only one meaning.
2) Input : Each algorithm should have zero or more inputs. Range of inputs for which algorithm works should be satisfied.
3) Output : Algorithm should produce correct results. Atleast one output has to be produced
4) Definiteness : Each instruction should be clear and unambiguous.
5) Effectiveness : Instruction should be simple and should transform the given input to desired output.
6) Finiteness :- Algorithm must terminate after a finite sequence of instruction.

Algorithm Specification :-

We can describe an algorithm in many ways.

- We can use natural language like english.
- We can make use of pseudocode
- We can make use of graphical representations called flowchart
- We must make sure that the resulting instructions are definite.

Steps for writing an algorithm :-

1) An algorithm is a procedure. It has two parts; the first part is head and the second part is body.
2) Head section consists of keyword 'Algorithm' and name of the algorithm with parameter list.

Eg :- Algorithm name {p1, p2 . . . pn}

The head section also has the following

// Problem description
// Input
// output

3) In the body of an algorithm various programming constructs like if, for, while and some statements like assignments are used.
4) The compound statements may be enclosed with { and } brackets. if, for, while can be closed by endif, endfor, endwhile respectively. Proper indention is must for block.
5) Comments are written using // at the beginning
6) The identifier should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.
7) The left arrow "←" used as assignment operator
Eg:- V←10
8) Boolean operators (TRUE, FALSE), Logical operators (AND, OR, NOT) and Relational operators (<, <=, >, >=, =, ≠, <>) are also used.
9) Input and Output can be done using read and write
10) Array[], if then else condition, branch and loop can be also used in algorithm.

Example:-

- Definition of an algorithm can be explained by computing GCD of 2 numbers.
- Greatest Common Divisor (GCD) of 2 numbers m and n denoted by GCD (m, n) is defined as the largest integer that divides both m and n such that the remainder is zero.
- GCD of 2 numbers is defined only for positive integers and not defined for negative integers and floating point number

Eutlid's algorithm :-

- Euclid's algorithm is based on applying repeatedly the equality $gcd(m,n) = gcd(n, m \bmod n)$ where '$m \bmod n$' is the remainder of the division of 'm by n', until '$m \bmod n$' is equal to zero.

Eg :- GCD(6,10) can be computed as

$GCD(6,10) = GCD(10,6) = GCD(6,4) = GCD(4,2) = GCD(2,0)$

$\therefore GCD(6,10) = 2$

Euclid's algorithm for computing gcd(m,n) in simple steps:-

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.

Euclid's algorithm for computing gcd(m,n) expressed in pseudocode :-

```
ALGORITHM Euclid_gcd(m,n)
// computes gcd(m,n) by Euclid's algorithm
// Input: Two non-negative, not both zero integers m & n.
// output: Greatest common divisor of m & n.
  while n ≠ 0 do
      r ← m mod n
      m ← n
      n ← r
  end while
  return m
```

Fundamentals of algorithmic problem solving :-

Sequence of steps involved in designing and analyzing an algorithm is as shown in the figure below.

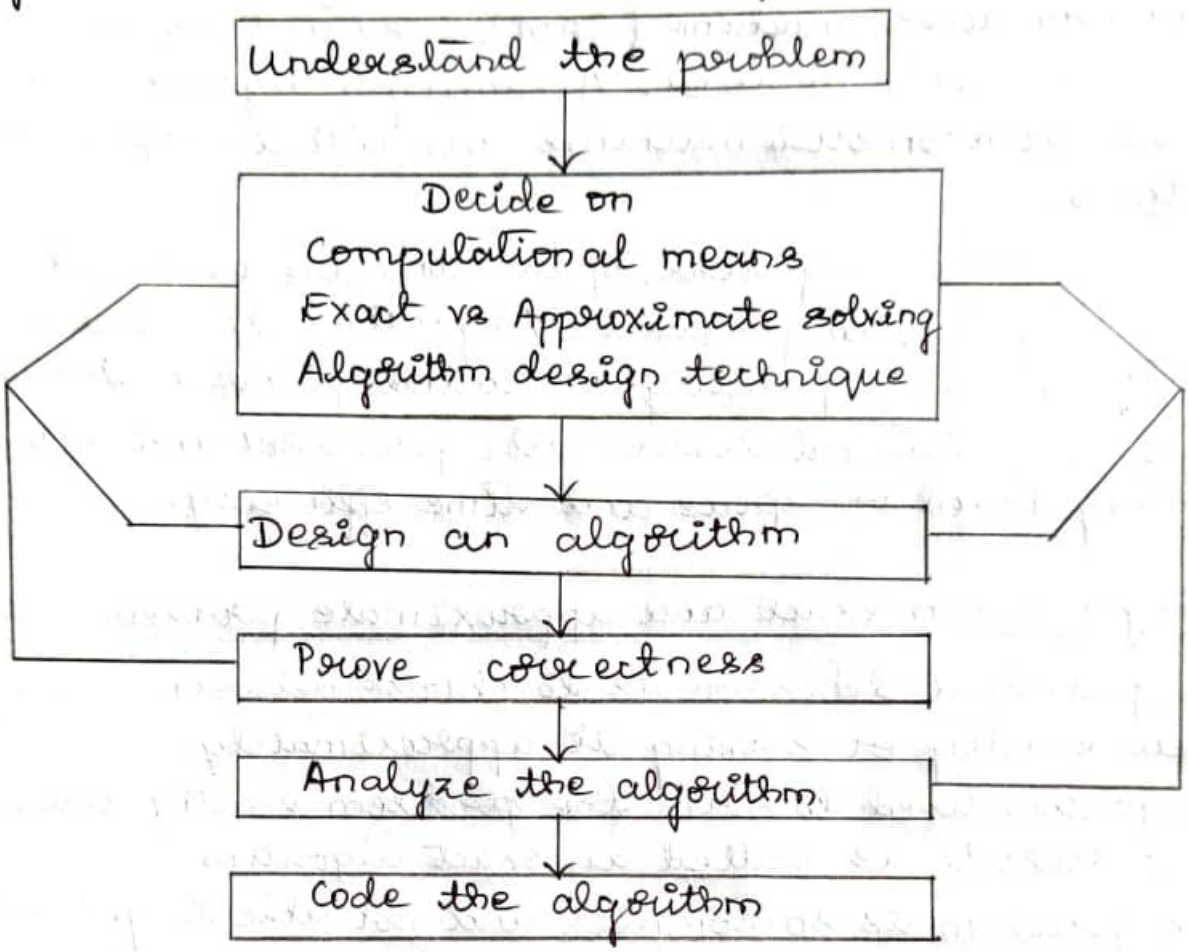


Fig :- Algorithm design and analysis process

1) Understanding the problem :-

- 1st 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
- An input to an algorithm specifies an instance of the problem the algorithm solves.
- Specify exactly the range of instances the algorithm needs to handle.

2) 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. 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 takes an advantage of this capability are called parallel algorithm.
- Choice of computational devices like processor and memory is mainly based on space and time efficiency.

b) Choosing between exact and approximate problem solving

- Next principal decision is to choose between solving the problem exactly or 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.

Eg :- Extracting square roots, solving non-linear equation

c) Algorithm design techniques :-

- An algorithm design technique 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 are a general approach by which many problems can be solved algorithmically.

Eg :- Brute force, divide and conquer, dynamic programming

3) Methods of specifying an algorithm :-

- Once we have designed an algorithm, we need to specify some fashion either in words or in pseudocode.
- Words and pseudocode are the 2 options that are most widely used nowadays for specifying algorithms.
- It is very simple and easy to specify an algorithm using natural language. But many times specification of an algorithm by using natural language is not clear
- Such specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of pseudocode.
- Pseudocode is a mixture of a natural language and a programming language constructs.
- Pseudocode is more precise than a natural language and its usage often yields more clear algorithm descriptions.

4) Proving an algorithm's correctness :-

- Once an algorithm has been specified, we have to prove its correctness.
- We have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- For some algorithms, a proof of correctness is quite easy, for others it can be quite complex.
- A common technique for proving correctness is to use mathematical induction because an algorithm's iteration

provide a natural sequence of steps needed for such proofs.

- In order to show that an algorithm is incorrect, we need just one instance of its input for which the algorithm fails.

5) Analyze the algorithm :-

- After correctness by far the most important is efficiency.
- There are 2 kinds of algorithm efficiency — Time efficiency and space efficiency
- Time efficiency indicates how fast the algorithm runs
- Space efficiency indicates how much extra memory the algorithm needs.
- The efficiency of an algorithm is determined by measuring both time and space efficiency.
- Factors to analyze an algorithm are
    - Time efficiency of an algorithm
    - Space efficiency of an algorithm
    - Simplicity of an algorithm
    - Generality of an algorithm.

6) Coding an algorithm :-

- Coding / implementation of an algorithm is done by a suitable programming language like C, C++ or JAVA.
- 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 be reduced by inefficient implementation.
- It is very essential to write an optimized code / efficient code to reduce the burden of compiler.

Analysis Framework :-

- Main purpose of algorithm analysis is to design most efficient algorithms
- The efficiency of an algorithm depends on 2 factors — Space and time efficiency.

Space efficiency :-

- Space efficiency of an algorithm is the amount of memory required to run the program completely and efficiently
- Space complexity of an algorithm depends on the factors
  - program space
  - Data space
  - Stack space

Program space :- It is the space required for storing the machine program generated by the compiler.

Data space :- It is the space required to store the constant variables etc

Stack space :- It is the space required to store the return address along with parameters that are passed to function, local variables etc

Time efficiency :-

- Time efficiency of an algorithm is measured purely on how fast a given algorithm is executed.
- Time efficiency of the algorithm depends on various factors such as
  - Speed of the computer
  - Choice of the programming language
  - Compiler used
  - Choice of the algorithm
  - Size of input and output

- Time efficiency of an algorithm depends on size of input 'n' and hence the time efficiency is always expressed in terms of 'n'
- Value of 'n' is directly proportional to the size of data to be processed, because all the algorithm runs longer on larger input.

Algorithms analysis framework consists of the following
1) Measuring an input's size
2) Unit for measuring running time
3) Orders of growth
4) worst-case, Best-case and Average-case efficiencies.

1) 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 straight forward.

Eg :- It will be the size of the list for problems of sorting and searching.

Eg :- For the problem of evaluating a polynomial $p(x) = a_n x_n + \cdots + a_0$ of degree 'n', size of parameter will be the polynomial's degree or the no of its co-efficients which is larger by 1 than its degree.

- There are situations where the choice of a parameter indicating an input size does matter.

Eg :- Computing the product of 2 $n \times n$ matrix

- Consider a spell checking algorithm. If the algorithm examine individual characters of its input then we should measure the size by the number of characters.
- In measuring input size for algorithms solving problems such as checking prime number of a positive integer 'n'

the input is just one number.

- Input size by the number 'b' of bits in the n's binary representation is

$$b = (\log_2 n) + 1$$

2) Units for measuring running time :-

- Some standard unit of time measurement such as a second millisecond and so on can be used to measure the running time of a program after implementing the algorithm.
- There are drawbacks to such an approach
  a) Dependence on the speed of a particular computer
  b) Dependence on the quality of a program implementing the algorithm
  c) The compiler used in generating the machine code
  d) The difficulty of checking the actual running time of the program.
- So, we need metric to measure an algorithm's efficiency that does not depend on these factors.
- one possible approach is to count the number of times each of the algorithm's operation is executed.
- Identify the most important operation of the algorithm called the basic operation
- Established frame work for the analysis of an algorithm time efficiency suggests measuring the basic operation by counting the number of times the algorithm's basic operation is executed on input of size 'n'.

Eg :-
Let C_{op} → be the execution time of an algorithm's basic operation
$C(n)$ → number of times this operation needs to be executed
we can estimate the running time $T(n)$ of a program implementing the algorithm by

$$T(n) \approx C_{op} . C(n)$$

- The count $C(n)$ does not contain any information about operation that are not basic and count itself is often computed approximately.

Basic Operation :-

- Statement that executes maximum number of times in a function is called basic operation.
- Number of times basic operation is executed depends on size of the input.
- Basic operation is the most time consuming operation in the algorithm.

Eg :- Statement present in the innermost loop in the algorithm.

3) Order of growth :-

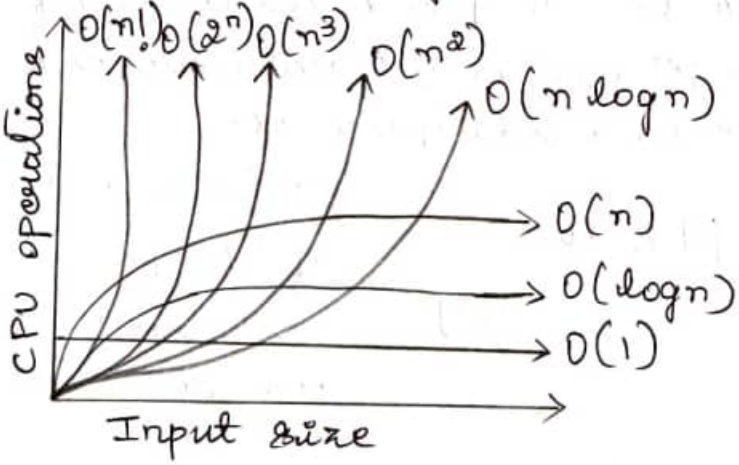
- We expect the algorithms to work faster for all values of 'n'
- Some algorithms execute faster for smaller values of 'n'. But as the value of 'n' increases, they tend to be very slow
- So, the behaviour of some algorithm changes with increase in value of 'n'.
- This change in behavior of the algorithm and algorithm's efficiency can be analyzed by considering the highest order of 'n'.
- Order of growth is normally determined for larger values of 'n' and it is used to measure the performance of an algorithm based on the input size.
- For large values of input size 'n', function order of growth counts.

Common computing time function used in order of growth are as follows:

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	$2.0 \cdot 10^7$	10^{12}	10^{18}		

- By comparing 'n' which is linear and 2^n which is exponential from the above table, we can say that exponential function grows very fast even for small variation of 'n'.
- So, algorithm with linear running time is preferred over exponential running time.
- All the function can be ordered according to their order of growth as shown from lowest to highest.

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



- Linear value is having constant value
- Function growing the slowest among these is the logarithmic function
- 2^n and $n!$ both these function grows fast that their values become large even for small values of 'n'.

- 2^n and $n!$ are known as exponential growth function
- For a very large value of 'n', exponential function generate a very high value such that even a fastest computer can take years to execute an algorithm.

Eg :- Computer which execute 10^{12} instruction / second takes $4 * 10^{10}$ years to execute 2^{100} operation.

4) Worst-case, Best-case and Average case efficiencies :-

- For some of the problems, time complexity will not depend on the no of inputs alone. It also depends on the efficiences of the algorithm.
- Consider as an example, sequential search. It is a straight forward algorithm that searches for a given item in a list of 'n' elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

ALGORITHM Sequential search (A[], K)
// Searches for a given value in a given array by sequential search
// Input : An array A[0 ... n-1] and search key K
// output : Index of the 1st element of A that matches K or -1 if there are no matching elements.

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i+1
if i < n return i
else return -1
```

[or]

```
for i ← 0 to n-1 do
    if (A[i] == Key) then
        return i
end for
```

Worst-case efficiency :-

- The efficiency of an algorithm for the input of size 'n' for which the algorithm takes longest time to execute among all possible input is called worst case efficiency.

Eg :- In sequential search, if the element to be searched is not present in the array, then it is a worst case efficiency.

A [10, 20, 30, 40, 50] $n = 5$ Key = 50

A[i] == Key
A[0] =/= 50
A[1] =/= 50
A[2] =/= 50
A[3] =/= 50
A[4] == 50

Here, search is successful after comparing all the elements in the list.
Searching element is found after comparing $n = 5$ elements, so it is worst case efficiency and it is denoted by

$$C_{worst}(n) = n$$

Best - Case efficiency :-

- The efficiency of an algorithm for the input of size for which the algorithm takes least time during execution among all the possible input of that size is called best case efficiency.

Eg :- In sequential search, if the element to be searched is present in the beginning, then it is a best case efficiency.

A [10, 20, 30, 40, 50] $n = 5$ Key = 10

A [i] == Key
A [0] == 10

Search is successful for single comparison so it is best case efficiency and it is denoted by

$$C_{best}(n) = 1$$

Average - case efficiency :-

- The efficiency of an algorithm for the input of size 'n' for which the algorithm takes average time to execute among all possible input, is called average - case efficiency.

Eg :- In sequential search, if the element to be searched is present somewhere in the middle of the array then it is a average case efficiency.

- Since, we don't know where the element is we have to consider the average number of cases.
- The standard assumptions are

a) The probability of a successful search is = 'p' $(0 \le p \le 1)$

b) Probability of the 1st match occurring in the i^{th} position of the list is the same for every 'i'

- Under these assumptions, we can find the average number of key comparisons $C_{avg}(n)$ as follows

In the case of a successful search,

$\frac{p}{n}$ → probability of the first match occurring in the i^{th} position of the list for every 'i'

i → Number of comparisons made by the algorithm in such situation is 'i'

In the case of a unsuccessful search

$1-p$ → Number of comparisons is 'n' with the probability of unsuccessful search.

$C_{avg}(n)$ = Probability of successful search + Probability of unsuccessful search

$$= \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1-p)$$

$$= \frac{p}{n}\left[1 + 2 + \cdots + i + \cdots + n\right] + n(1-p)$$

$$= \frac{p}{n} \frac{n(n+1)}{2} + n(1-p)$$

$$C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$$

where, $1 \cdot \frac{p}{n}$ → successful search at 1st position

$2 \cdot \frac{p}{n}$ → successful search at 2nd position and so on.

Asymptotic Notations and Basic Efficiency classes :-

- The efficiency of the algorithm is normally expressed using order of growth of an algorithms basic operation count as the principal indicator of the algorithm's efficiency.
- Order of growth can be expressed using 2 methods
  1) Order of growth using asymptotic notation
  2) Order of growth using limits.

Asymptotic notation definition :-

- The value of the function may increase/decrease as the value of 'n' increases. Based on the order of growth of n, the behaviour of the function varies.
- Asymptotic notation are the notation using which 2 algorithms can be compared with respect to efficiency based on the order of growth of an algorithms basic operation.

Types of asymptotic notation :-

1) O (Big Oh)
2) Ω (Big omega)
3) θ (Big Thela)

Informal definition :-

1) Big - oh (O) notation :-

- Assuming 'n' indicates the size of input and g(n) is a function, informally O(g(n)) is defined as set of function with a small or same order of growth as g(n) as 'n' goes to infinity.

Eg :- Let $g(n) = n^3$

Since n & n^2 have smaller order of growth and n^3 has same order of growth, we say

$n \in O(n^3)$ $\quad n^3 \in O(n^3)$

$n^2 \in O(n^3)$

Eg :- Let $g(n) = n^3$

Since n^4, $0.001n^4$, n^5+n+3 do not have smaller order or same order as that of n^3, they do not belong to $O(g(n))$ and can be represented as

$n^4 \notin O(n^3)$ $\quad 0.001n^4 \notin (n^3)$ $\quad n^5+n+3 \notin O(n^3)$

2) Big omega (Ω) notation :-

- Assuming 'n' indicates the size of input and $g(n)$ is a function informally $\Omega(g(n))$ is defined as set of function with a larger or same order of growth as $g(n)$ as 'n' goes to infinity.

Eg :- Let $g(n) = n^3$

Since n^3, n^4, $0.001n^4$, n^5+n+3 have larger order or same order as that of n^3. Hence, they belong to $\Omega(g(n))$ and we can say

$n^3 \in \Omega(n^3)$ $\quad n^4 \in \Omega(n^3)$ $\quad 0.001n^4 \in \Omega(n^3)$

$n^5+n+3 \in \Omega(n^3)$

Eg :- Let $g(n) = n^3$

Since n & n^2 do not have larger order or same order of growth when compared to n^3, we can say

$n \notin \Omega(n^3)$ $\quad n^2 \notin \Omega(n^3)$

3) Big Thela (θ) notation :-

- Assuming 'n' indicates the size of input and $g(n)$ is a function, informally $\theta(g(n))$ is defined as set of function that have same order of growth as $g(n)$ as 'n' goes to infinity.

Eg :- $g(n) = n^2$

Since an^2+bn+c for $a>0$, $n^2+\sin n$, $n^2+\log n$ have same order as n^2. Hence, they belong to $\theta(g(n))$

$an^2+bn+c \in \theta(n^2)$ $\quad n^2+\sin n \in \theta(n^2)$ $\quad n^2+\log n \in \theta(n^2)$

Formal definition :-

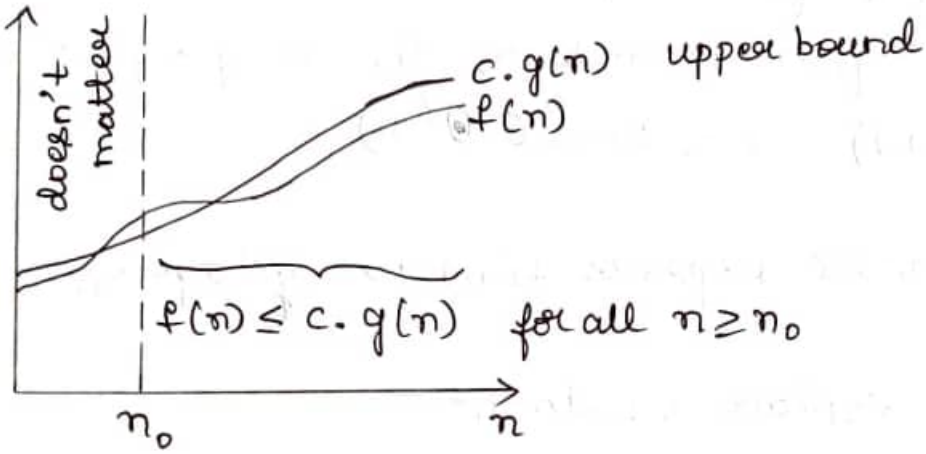
1) Big oh (O) notation :-

- Let $f(n)$ be the time efficiency of an algorithm. The function $f(n)$ is said to be $O(g(n))$, denoted by

$$f(n) \in O(g(n)) \quad (\text{or}) \quad f(n) = O(g(n))$$

if and only if there exists a positive constant 'c' and positive integer n_0, satisfying the constraint.

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



- Here, $c.g(n)$ is the upper bound. The upper bound on $f(n)$ indicates that function $f(n)$ will not consume more than specified time $c.g(n)$.
- Big-oh notation is used for finding worst-case time efficiency.
- Running time of function $f(n)$ may be equal to $c.g(n)$, but it will never be worse than the upper bound.
- So, $f(n)$ is generally faster than $g(n)$

Note :-

Big-oh is the formal method of expressing the upper bound of an algorithm running time. It is a measure of the longest amount of time it could possibly take for the algorithm to complete.

Eg:- Let $f(n) = 100n + 5$. Express $f(n)$ using big-oh.

Sol^n:-

$f(n) = 100n + 5$. Replacing 5 with n (so that next higher order term is obtained)

we get, $100n + n = 101n \rightarrow C.g(n)$

i.e, $C.g(n) = 100n + n = 101n$ for $n = 5$

Now, the constraint is satisfied.

$f(n) \leq C.g(n)$ for $n \geq n_0$

i.e, $100n + 5 \leq 101n$ for $n \geq 5$

where $C = 101$ $g(n) = n$ and $n_0 = 5$. So by definition

$f(n) \in O(g(n))$ i.e, $f(n) \in O(n)$

Eg:- Let $f(n) = 10n^3 + 8$. Express $f(n)$ using big-oh.

Sol^n:-

$f(n) = 10n^3 + 8$. Replace 8 with n^3

we get, $10n^3 + n^3 = 11n^3 \rightarrow C.g(n)$

i.e, $C.g(n) = 10n^3 + n^3 = 11n^3$ for $n = 8$

Now, the constraint is satisfied

$f(n) \leq C.g(n)$ for $n \geq n_0$

i.e, $10n^3 + 8 \leq 11n^3$ for $n \geq 8$

where $C = 11$ $g(n) = n^3$ and $n_0 = 8$. So by definition

$f(n) \in O(g(n))$ i.e, $f(n) \in O(n^3)$

Note:-

Big-oh notation is widely used because, we normally take worst case scenario & prepare for the worst. The only limitation of big-oh is that there is no lower bound for $f(n)$ for large value of 'n'.

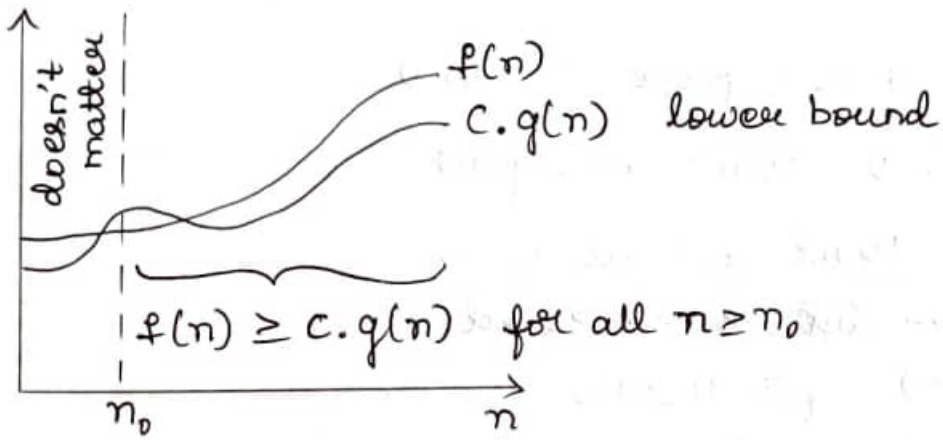
2) Big-Omega (Ω) notation :-

- Let $f(n)$ be the time complexity of an algorithm. The function $f(n)$ is said to be $\Omega(g(n))$ 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 a positive integer n_0 satisfying the constraint

  $f(n) \geq c * g(n)$ for all $n \geq n_0$



- This notation gives the lower bound on a function $f(n)$ within a constant factor
- Lower bound on $f(n)$ indicates that function $f(n)$ will consume at least the specified time $C.g(n)$
  i.e, the algorithm has a running time that is always greater than $C.g(n)$.
- Lower bound implies that below this time the algorithm cannot perform better.
- As the running time $f(n)$ of an algorithm is always greater than $g(n)$, big-omega notation is used for finding best case time efficiency.

Eg :- $f(n) = 100n+5$. Express $f(n)$ using big-omega.

Sol^n :- $f(n) = 100n+5$. Replace 5 with 0 (so that lowest order term is obtained)

we get, $100n+0 = 100n \rightarrow C.g(n)$

i.e, $C.g(n) = 100n$ for $n=0$

Now, the constraint is satisfied

$f(n) \geq c.g(n)$ for $n \geq n_0$

i.e, $100n + 5 \geq 100n$ for $n \geq 0$

where $c = 100$ $g(n) = n$ and $n_0 = 0$. So, by definition

$f(n) \in \Omega(g(n))$ i.e, $f(n) = \Omega(n)$

Eg :- $f(n) = 10n^3 + 5$. Express $f(n)$ using big-omega.

Soln :- $f(n) = 10n^3 + 5$. Replace 5 with 0

we get, $10n^3 + 0 = 10n^3 \rightarrow c.g(n)$

i.e, $c.g(n) = 10n^3$ for all $n = 0$

Now, the constraint is satisfied

$f(n) \geq c.g(n)$ for $n \geq n_0$

i.e, $10n^3 + 5 \geq 10n^3$ for $n \geq 0$

where $c = 10$ $g(n) = n^3$ and $n_0 = 0$. So, by definition

$f(n) \in \Omega(g(n))$ i.e, $f(n) = \Omega(n^3)$.

3) Big-Theta (θ) notation :-

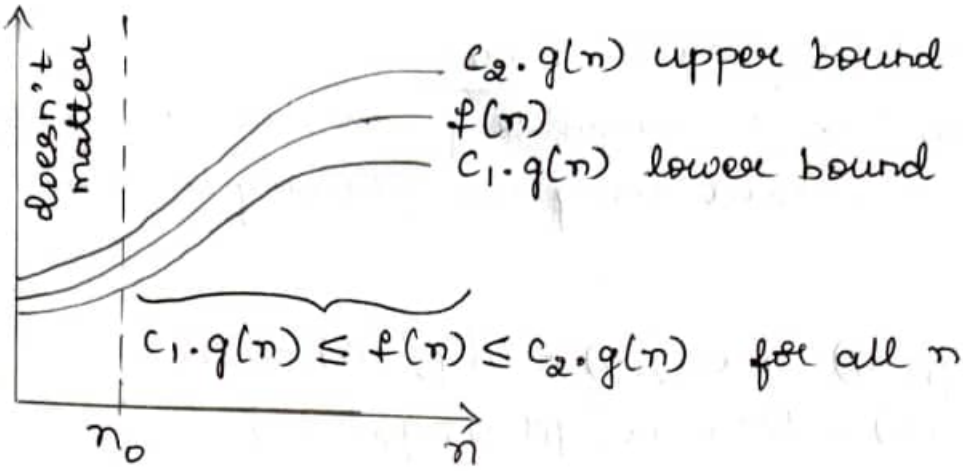
- Let $f(n)$ be the time complexity of an algorithm. The function $f(n)$ is said to be big-theta of $g(n)$ denoted by

  $f(n) \in \theta(g(n))$ (or) $f(n) = \theta(g(n))$

  if and only if there exists some positive constants 'c_1' and 'c_2' and positive integer n_0 satisfying the constraint

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

- This notation is used to denote both lower bound and upper bound on a function $f(n)$ within a constant factor.
- Upper bound on $f(n)$ indicates that function $f(n)$ will not consume more than the specified time $c_2.g(n)$.



- Lower bound on $f(n)$ indicates that function $f(n)$ in the best case will consume atleast the specified time $c_1 \cdot g(n)$.

Eg :- Let $f(n) = 100n + 5$. Express $f(n)$ using big-theta.

Soln :-

The constraint to be satisfied is

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

$100n \leq 100n + 5 \leq 105n$ for all $n \geq 1$

where $c_1 = 100$, $c_2 = 101$ and $n_0 = 1$ $g(n) = n$. So, by definition

$f(n) \in \theta(g(n))$ i.e, $f(n) \in \theta(n)$

Eg :- Let $f(n) = 10n^3 + 5$. Express $f(n)$ using big-omega.

Soln :-

The constraint to be satisfied is

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

$10n^3 \leq 10n^3 + 5 \leq 11n^3$ for all $n \geq 2$

$n^3 \geq 5$

i.e, $n \geq 2$

where $c_1 = 10$, $c_2 = 11$, $n_0 = 2$ and $g(n) = n^3$. So by definition

$f(n) \in \theta(g(n))$ i.e, $f(n) \in \theta(n^3)$

Property of an asymptotic notation :-

- If an algorithm has 2 executable parts, the analysis of this algorithm can be obtained using the following theorem

Theorem :-

If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then
$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

Proof :-

- By definition, we know that the function $f(n)$ is said to be big-oh (O) of $g(n)$ denoted by
$f(n) \in O(g(n))$
such that there exists a positive constraint 'c' and positive integer n_0 satisfying the constraint
$f(n) \leq c.g(n)$ for all $n \geq n_0$
- It is given that $f_1(n) \in O(g_1(n))$, so by definition there exists a relation $f_1(n) \leq c_1.g_1(n)$ for $n \geq n_1$ — ①
- It is given that $f_2(n) \in O(g_2(n))$, so by definition there exists a relation $f_2(n) \leq c_2.g_2(n)$ for $n \geq n_2$ — ②
- Assume $c_3 = \max\{c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$ — ③
By adding eq^n ① and ②

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1.g_1(n) + c_2.g_2(n) \\ &\leq c_3.g_1(n) + c_3.g_2(n) \quad \text{from eq}^n\ ③ \\ &\leq c_3[g_1(n) + g_2(n)] \end{aligned}$$

By mathematical rule, if a_1, a_2, b_1 & b_2 are 4 terms and if $a_1 < b_1$, $a_2 < b_2$ then $a_1 + a_2 \leq 2\max\{b_1, b_2\}$

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_3[g_1(n) + g_2(n)] \\ &\leq c_3\, 2 \max\{g_1(n), g_2(n)\} \end{aligned}$$

where $a_1 = f_1(n)$, $a_2 = f_2(n)$, $b_1 = g_1(n)$, $b_2 = g_2(n)$

Since, $f_1(n) + f_2(n) \leq 2c_3 \max\{g_1(n), g_2(n)\}$ then by definition, we can write

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

where,

$C = 2C_3 = 2\max\{C_1, C_2\}$ and

$n_0 = \max\{n_1, n_2\}$. Hence the proof.

- It is clear from the above property that, the overall efficiency of the algorithm is determined by the executable part which is larger order of growth.

Order of growth using limits :-

- The order of growth can be obtained by computing the limit of the ratio of the 2 function as shown below

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{imply } f(n) \text{ has smaller order of growth than } g(n) \text{ i.e, } f(n) \in O(g(n)) \\ C & \text{imply } f(n) \text{ has same order of growth than } g(n) \text{ i.e, } f(n) \in \Theta(g(n)) \\ \infty & \text{imply } f(n) \text{ has larger order of growth than } g(n) \text{ i.e, } f(n) \in \Omega(g(n)) \end{cases}$$

- To compute the order of growth, the limit based approach is often convenient because, it is more advantage to use powerful calculus techniques such as

1) L' Hospital's rule defined by

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

2) Stirling's formula which is given by

$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for very large values of 'n'.

Examples :-

1) Compare order of growth $\frac{1}{2}n(n-1)$ and n^2 using limits.

Sol^n :- $f(n) = \frac{1}{2}n(n-1)$ $\quad g(n) = n^2$

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

$$= \frac{1}{2}\lim_{n\to\infty} \frac{n(n-1)}{n^2} \Rightarrow \frac{1}{2}\lim_{n\to\infty}\left[\frac{n^2-n}{n^2}\right]$$

$$= \frac{1}{2}\lim_{n\to\infty}\left[\frac{n^2}{n^2} - \frac{n}{n^2}\right] \Rightarrow \frac{1}{2}\lim_{n\to\infty}\left[1 - \frac{1}{n}\right]$$

Applying limit ∞

$$= \frac{1}{2}\left[1 - \frac{1}{\infty}\right] \Rightarrow \frac{1}{2}[1-0] = \frac{1}{2}$$

$\therefore \lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{1}{2}$ i.e, constant 'c'

Since $\frac{1}{2}$ is a positive constant, both the function $f(n)$ and $g(n)$ have the same order of growth and it is represented as

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

i.e, $\frac{1}{2}n(n-1) \in \theta(n^2)$

2) Compare order of growth of $\log_2 n$ and $\sqrt{n}$.

Sol^n :- $f(n) = \log_2 n$ $\quad g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}}$$

Applying L' Hospital's rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

$$= \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'}$$

$$\lim_{n\to\infty} (\log_2 n)' = (\log_2 e)\frac{1}{n}$$

$$\lim_{n\to\infty} (\sqrt{n})' = \frac{1}{2\sqrt{n}}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} \Rightarrow \lim_{n\to\infty} 2\sqrt{n}\,(\log_2 e)\frac{1}{n}$$

$$= 2\log_2 e \lim_{n\to\infty} \frac{\sqrt{n}}{n} \Rightarrow 2\log_2 e\left(\frac{\sqrt{\infty}}{\infty}\right)$$

$$= 2\log_2 e\,(0) = 0$$

$$\therefore \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Since the limit is equal to 0, the $f(n) = \log_2 n$ has a smaller order of growth than $g(n) = \sqrt{n}$ and it is denoted as

$$\log_2 n \in O(\sqrt{n}) \quad \text{i.e, } f(n) \in O(g(n))$$

3) Compare the order of growth of $n!$ and 2^n

Sol^n:-

$f(n) = n!$ and $g(n) = 2^n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n!}{2^n}$$

Using sterling's formula:- $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}$$

$$= \lim_{n\to\infty} \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n \cdot \frac{1}{2^n} \Rightarrow \lim_{n\to\infty} \sqrt{2\pi n}\cdot\frac{n^n}{e^n\cdot 2^n}$$

$$= \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n \Rightarrow \sqrt{2\pi\infty}\left(\frac{\infty}{2e}\right)^\infty \Rightarrow \infty$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Since the limit is equal to ∞, the $f(n)$ has a larger order of growth than $g(n)$ and it is represented as

$n! \in \Omega(2^n)$ i.e, $f(n) \in \Omega(g(n))$

Basic Efficiency of classes:-

- Time efficiences of a large number of algorithms include only few classes
- Efficiency of classes are used to represent asymptotic notations of order of growth.
- Classes are listed below in the increasing order of their orders of growth.

1) 1 → Constant
   - Indicates that running time of a program is constant
   - Specifies best case efficiency

2) log n → logarithmic
   - This running time occurs in programs that solve larger problem by reducing the problem size by constant factor at each iteration of the loop.
   Eg :- Binary search

3) n → linear
   - Occurs in algorithm that scan a list of size 'n'
   Eg :- Sequential search

4) n log n
   - Occurs in algorithm to sort elements in ascending order such as quick sort, merge sort etc i.e, divide & conquer algorithm

5) n^2 → quadratic
   - Algorithm have 2 loops such as sorting algorithm

Eg :- Bubble sort, selection sort, addition and multiplication of 2 matrices.

6) $n^3 \rightarrow$ cubic

- Algorithm have 3 loops

Eg :- Matrix multiplication

7) $2^n \rightarrow$ Exponential

- Occurs in algorithm that generates subset of a given set

Eg :- Tower of Hanoi

8) $n! \rightarrow$ Factorial

- Occurs in algorithm that generate all permutation of set.

Mathematical analysis of non-recursive algorithms :-

General plan for analyzing non-recursive algorithm :-

1) Based on the input size, determine the number of parameters to be considered
2) Identify the algorithm's basic operation
3) Check whether the number of times the basic operation is executed depends only on the size of input.
   If the basic operation to be executed depends on some other condition then it is necessary to obtain the worst, best and average case separately.
4) Obtain the total number of times a basic operation is executed
5) Simplify using standard formulas and obtain the order of growth

Basic rules of sum manipulation :-

$$\sum_{i=l}^{u} c a_i = c \sum_{i=l}^{u} a_i$$

$l \rightarrow$ lower limit
$u \rightarrow$ upper limit

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

Summation formulas :-

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \leq u$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n$$

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

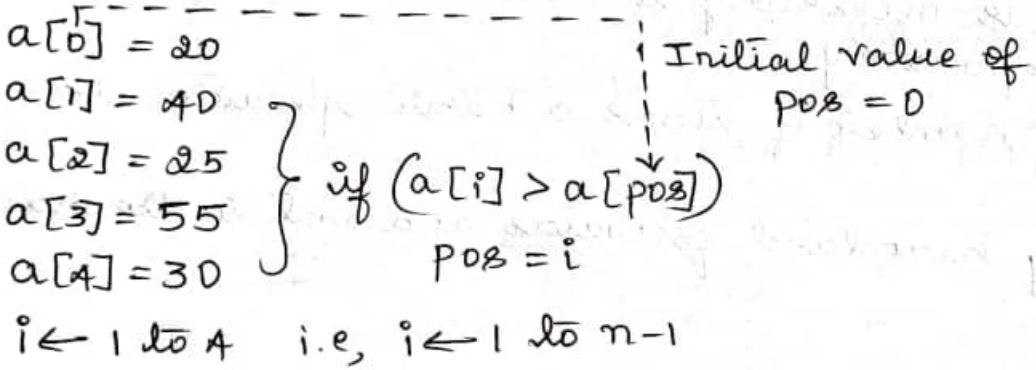
Consider some of the algorithms and see how to analyze these algorithms.

1) Maximum of 'n' elements :-

Design the algorithm to find largest of 'n' numbers and obtain the time efficiency.

Design :-

- Consider the array 'a' consisting of the 5 elements such as 20, 40, 25, 55 and 30.
- Here $n = 5$ represent the number of elements in the array
- Parameters are 'a' and 'n'. The pictorial representation is shown below



Algorithm :-

```
Algorithm Maximum (a[], n)
// Find the largest of 'n' number
// Input : n — number of elements in array
           a — array consisting of 'n' elements
// Output : pos — contains the position of largest elements
```

```
pos ← 0
for i ← 1 to n-1 do
if (a[i] > a[pos])
   pos ← i
end for
return pos
```

Analysis :-

Time efficiency can be calculated as

Step-1 :- parameter to be considered is n – input size

Step-2 :- Basic operation is element comparison

i.e, if $(a[i] > a[pos])$

Step-3 :- Total number of times the basic operation is executed can be calculated as

$f(n) \rightarrow$ Number of times comparison is executed

Algorithm makes 1 comparison on each execution of the loop, which is repeated for each value of the loop's variable 'i' within the bounds 1 and $n-1$.

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

In general, $f(n) = \sum_{i=l}^{u} 1$ then Result $= u - l + 1$

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

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

$$= n - 1 \approx n$$

$$f(n) = n$$

Step-4 :- Express $f(n)$ using asymptotic notation. So, time complexity is given by

$$f(n) \in \theta(n)$$

2) Matrix Multiplication :-

Given 2 $n \times n$ matrices a & b, find their time efficiency for computing product $c = ab$

Design :-

Two matrices a and b can be multiplied and the result can be stored in matrix c for each value of i and j as shown below

$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j] \quad \text{for } i = 0 \text{ to } n-1 \text{ and for } j = 0 \text{ to } n-1$$

Algorithm :-

```
Algorithm multiplication (a[], b[], c[], n)
// Multiply 2 matrices a & b of size n x n
//Input : n → represent size of array
          a → 1st matrix of size n x n
          b → 2nd matrix of size n x n
//Output : c → Resultant matrix where the product of 2 matrices
           should be stored.
  for i ← 0 to n-1
      for j ← 0 to n-1
          sum ← 0
          for k ← 0 to n-1
              sum ← sum + a[i][k] * b[k][j]
          end for
      c[i][j] ← sum
  end for
end for
```

Analysis :-

Time complexity can be calculated as follows

Step-1 :- The parameter to be considered is 'n' → input size

Step-2 :- Basic operation is the multiplication statement

i.e, sum ← sum + a[i][k] * b[k][j]

Even though addition operation is present, we do not have to

- Choose between these 2 operation, because on each repetition of the loop both the operation is executed only once. So, by counting one we automatically count the other.
- Since the count depends on the value of 'n' and not on any other factors. So, the total number of times the multiplication statement is executed can be obtained as shown
- As the count depends only on the input, we do not have to calculate worst and best case separately.

Step-3 :- Number of multiplication depends on the value of 'n' and not on any other factors.

```
for i = 0 to n-1
    for j ← 0 to n-1
        sum ← 0
        for k ← 0 to n-1
            sum ← sum + a[i][k] * b[k][j]
```

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

By considering, $f(n) = \sum_{i=l}^{u} 1$ Result $= u - l + 1$

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (n-1) - 0 + 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= \sum_{i=0}^{n-1} n \sum_{j=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} n \left[(n-1) - 0 + 1\right]$$

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

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

$$= n^2[(n-1) - 0 + 1]$$

$$= n^2(n)$$

$$f(n) = n^3$$

$f(n) \in \Theta(n^3) \Rightarrow$ Time complexity

3) Element Uniqueness Problem :-

Let us design an algorithm to check whether all the elements in a given array are distinct and find the time efficiency. Consider the following 2 arrays.

a[0] = 10 a[1] = 20 a[2] = 30 a[3] = 40 a[4] = 50	Elements are Unique
a[0] = 10 a[1] = 20 a[2] = 10 a[3] = 40 a[4] = 50	Elements are not unique.

Design :-

Consider an array with 5 elements. For the 1st time item a[0] should be compared with a[1], a[2], a[3], a[4]. 2nd time, item a[1] should be compared with a[2], a[3], a[4]. 3rd time, item a[2] should be compared with a[3], a[4] and finally, a[3] should be compared with a[4]. Above procedure can be shown below

1st time	2nd time	3rd time	4th time
a[0] - a[1]	a[1] - a[2]	a[2] - a[3]	a[3] - a[4]
a[0] - a[2]	a[1] - a[3]	a[2] - a[4]	
a[0] - a[3]	a[1] - a[4]		
a[0] - a[4]			
i j	i j	i j	i j

From the above table

- Index variable 'i' starts from 0 and goes upto 3 as shown in dotted lines.

  $i \leftarrow 0$ to 3

  $i \leftarrow 0$ to $n-2$ — ①

- For each initial value of 'i' the index variable 'j' always starts from i+1 and goes upto 4.

  $j \leftarrow i+1$ to 4

  $j \leftarrow i+1$ to $n-1$ — ②

- If a[i] is same as a[j], then duplicate item is found and return 0

  if (a[i] = a[j]) return 0 — ③

Algorithm:-

```
Algorithm Unique_Elements (a[], n)
// Checks whether the elements in a given array are distinct
// Input: n → number of elements in array
          a → array consisting of n elements
// output: return 0 if elements are not unique
           return 1 if elements are unique.

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:-

The time efficiency of this algorithm can be computed as shown.

Step-1:- The parameter to be considered is 'n' → input size

Step-2 :- Basic operation is the statement

if (a[i] = a[j]) return 0

Step-3 :- Number of comparisons not only depends on the value of 'n' but depends on the position of repeated elements if present. The repeated elements if not present, the number of comparison are more. So, we need to find worst case and best case time efficiency.

Worst-Case efficiency :-

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

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

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

By, $\sum_{i=l}^{u} a_i \pm b_i = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

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

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

Replace n by n-2 as the upper limit is n-2

$$= (n-1)(n-2-0+1) - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)\left[(n-1) - \frac{(n-2)}{2}\right] \Rightarrow (n-1)\left[\frac{2n-2-n+2}{2}\right]$$

$$= (n-1)\left(\frac{n}{2}\right) \Rightarrow \frac{n^2-n}{2} \approx n^2$$

$f(n) = n^2$

$f(n) \in O(n^2)$

Best-case efficiency :-

- Best case occurs whenever the value stored in a[0] is same as a[1]. It is observed that in the best case, basic operation is executed only once. So, the time complexity is

$$f(n) \in \Omega(1)$$

Mathematical analysis of recursive algorithms :-

General plan for analyzing time efficiency of recursive algorithm

1) Decide on a parameter indicating an input size
2) Identify the algorithm's basic operation
3) Check whether the number of times the basic operation is executed can vary on different input of the same size; if it can, worst case, average case and best case efficiencies must be calculated separately
4) Set up a recurrence relation, with an appropriate initial condition calculate the number of times the basic operation is executed.
5) Solve the recurrence relation and generate the order of growth of its solution and express using asymptotic notation.

1) Factorial of a number:-

Design :- How to compute factorial of 5 using recursion ?

we can compute 5! as shown

$5! = 5 * 4!$
$4! = 4 * 3!$
$3! = 3 * 2!$
$2! = 2 * 1!$
$1! = 1 * 0!$
$0! = 1$

General case
$n! = n * (n-1)!$

Base case
$n! = 1$ if $n = 0$

Backward substitution

$0! = 1$

$1! = 1 * 0! = 1$

$2! = 2 * 1! = 2$

$3! = 3 * 2! = 6$

$4! = 4 * 3! = 24$

$5! = 5 * 4! = 120$

Thus the recursive definition can be written as

$$n! = 1 \quad \text{if } n = 0$$
$$n! = n * (n-1)! \quad \text{otherwise}$$

Above definition can also be written as

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

Algorithm :-

```
Algorithm Fact(n)
// Function computes factorial of n
// Input : n → positive integer
// output : Factorial of n
    if (n==0) return 1
    else return n * Fact(n-1)
```

Analysis :-

Time efficiency of the algorithm to find the factorial of a number can be obtained as shown.

Step-1 :- The parameter to be considered is $n \rightarrow$ input size

Step-2 :- Basic operation is multiplication statement

$n * Fact(n-1)$

Step-3 :- Total number of multiplication can be obtained using the recurrence relation as shown

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + f(n-1) & \text{otherwise} \end{cases} \quad \text{—— (1)}$$

$f(0) = 0$ → no multiplication when $n = 0$

Calls stop when $n = 0$

Recurrence relation can be solved using repeated substitution as shown below

$1 + f(n-1)$

To multiply $f(n-1)$ by n ← → To compute $f(n-1)$

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

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

$= 2 + f(n-2)$

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

$= 3 + f(n-3)$

$= 4 + f(n-4)$

$\vdots$

$= i + f(n-i)$

$f(n) = 1 + f(n-1)$ — (2)

$f(n-1) = 1 + f(n-1-1)$ Replacing n by $n-1$ in (2)

$= 1 + f(n-2)$

$f(n-2) = 1 + f(n-2-1)$ Replacing n by $n-2$ in (2)

$= 1 + f(n-3)$

To get the initial condition $f(0) = 0$ let $i = n$

$f(n) = i + f(n-i)$

$= n + f(n-n)$

$= n + f(0)$ $f(0) = 0$ from (1)

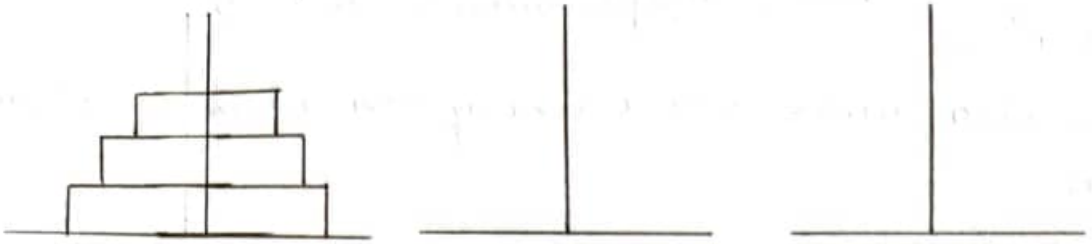
$f(n) = n$

$\therefore f(n) \in \theta(n)$

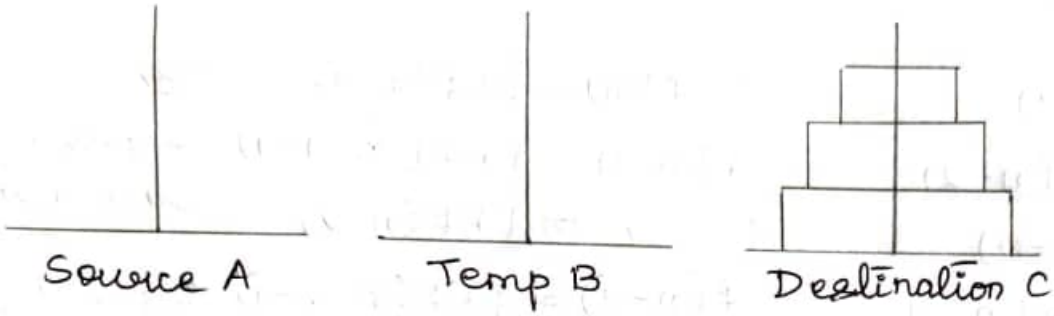
2) Towers of Hanoi :-

- In this problem, there are 3 needles say A, B, C. The different diameters of 'n' discs are placed one above the other through the needle 'A' and the discs are placed such that always a smaller disc is placed above the larger disc. 2 Needles B & C are empty. All the discs from needle A are to be transferred to needle C using needle B as temporary storage.

Initial setup:-



After transferring all discs from A to C we get



Algorithm :-

Algorithm Tower of Hanoi (n, source, temp, dest)
// To move 'n' disc from source to dest and see only 1 disc is moved at a time
// Input : n → total number of disks to be moved
// output : all 'n' disks should be on destination needle

Step-1 : Check for base case
if (n=0) return
Step-2 : Recursively move n-1 disk from source to temp
Tower of Hanoi (n-1, source, dest, temp)
Step-3 : Move n^{th} disk from source to dest
write (Move disk 'n' from source to dest)
Step-4 : Recursively move n-1 disk from temp to dest
Tower of Hanoi (n-1, temp, source, dest)

Analysis :-

Time efficiency of this algorithm can be calculated as shown

Step-1 :- Parameter to be considered is 'n' → number of disks to be moved.

Step-2 :- Basic operation is movement of disk

Step-3 :- Total number of disk movements can be obtained using the recurrence relation as

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ f(n-1) + 1 + f(n-1) & \text{otherwise} \end{cases}$$

- $f(n-1)$ ↙ Movement from source to temp
- 1 ↓ Movement from source to dest
- $f(n-1)$ ↘ Movement from temp to dest

Above recurrence relation can be written as

$$f(1) = 1 \quad \text{——— (1)}$$
$$f(n) = 2f(n-1) + 1$$

Recurrence relation can be solved using repeated substitution

$$f(n) = 2f(n-1) + 1 \quad \text{——— (2)}$$

$$f(n) = 2f(n-1) + 1$$
$$= 2[2f(n-2)+1] + 1$$
$$= 2^2 f(n-2) + 2 + 1$$
$$= 2^3 f(n-3) + 2^2 + 2 + 1$$
$$= 2^4 f(n-4) + 2^3 + 2^2 + 2 + 1$$
$$\vdots$$
$$= 2^i f(n-i) + \underbrace{2^3 + 2^2 + 2 + 1}$$

$f(n-1) = 2f(n-1-1)+1$
$= 2f(n-2)+1$ Replace n by $n-1$ in (2)

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

This geometric series can be solved using $S = \frac{a(r^n - 1)}{r-1}$

where $a = 1$, $r = 2$, $n = i$

↳ No of terms from 0 to $i-1 = i$

i.e, $2^{i-1} + 2^{i-2} - - - + 2^3 + 2^2 + 2 + 1$

$$S = \frac{a(r^n - 1)}{r-1} = \frac{1(2^i - 1)}{2-1}$$

$$S = 2^i - 1$$

Substitute this value in geometric series

$$f(n) = 2^i f(n-i) + 2^3 + 2^2 + 2 + 1$$
$$= 2^i f(n-i) + 2^i - 1$$

To get initial condition $f(1)$ let $i = n-1$

$$f(n) = 2^i f(n-i) + 2^i - 1$$

$$= 2^{n-1} f(n-(n-1)) + 2^{n-1} - 1$$

$$= 2^{n-1} f(n-n+1) + 2^{n-1} - 1$$

$$= 2^{n-1} f(1) + 2^{n-1} - 1 \qquad f(1) = 1 \text{ from eqn (1)}$$

$$= 2^{n-1} + 2^{n-1} - 1$$

$$= 2 \cdot 2^{n-1} - 1 \qquad 2^{n-1} = \frac{2^n}{2}$$

$$= 2 \cdot \frac{2^n}{2} - 1$$

$$f(n) = 2^n - 1$$

$$f(n) \in \Theta(2^n - 1) \in \Theta(2^n)$$

Note :- Time complexity of this algorithm is exponential in nature and so this algorithm should perform more computation even for a smaller value. This does not mean that the algorithm is poor. It is most efficient algorithm for this problem. Nature of the problem itself is such that it is computationally expensive.

3) Digits in a binary number of a given decimal number :-

How to find the number of digits of a binary corresponding to the decimal number ? It is not necessary to obtain the binary number.

Design :-

- Let us initial count to 1. As long as given N is greater than 1, keep dividing it by 2 and increment corresponding count by 1.
- Recursive definition to find the number of binary digits in a given decimal integer can be written as

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 1 + f(n/2) & \text{otherwise} \end{cases}$$

Algorithm :-

Algorithm Binary(n)
// To count the number of digits in a binary given positive decimal integer.
// Input : n → positive decimal integer
// output : number of binary digits in a given decimal integer

```
if (n == 1) return 1
else return 1 + Binary(n/2)
```

Analysis :-

Time efficiency can be computed as

Step-1 :- parameter to be considered is n → input size

Step-2 :- Basic operation is
1 + Binary(n/2) which will be executed recursively

Step-3 :- Total number of times basic operation is executed can be recursively found as

a) if n=1, basic operation is executed 0 times
$f(n) = 0$ if $n \leq 1$

b) if (n>1), basic operation is executed atleast once which is given by
$f(n) = 1 + f(n/2)$

So, the recurrence relation obtained in step-3 can be shown as.

$$f(n) = \begin{cases} 0 & \text{if } n \leq 1 \quad \text{---} (1) \\ 1 + f(n/2) & \text{otherwise} \end{cases}$$

Recurrence relation can be solved as

$f(n) = 1 + f(n/2)$
$= 1 + 1 + f\left(\frac{n}{2^2}\right)$
$= 2 + f\left(\frac{n}{2^2}\right)$

$f(n) = 1 + f(n/2)$ — (2)
$f\left(\frac{n}{2}\right) = 1 + f\left(\frac{n}{2^2}\right)$ Replacing n by $\frac{n}{2}$ in (2)
$f\left(\frac{n}{2^2}\right) = 1 + f\left(\frac{n}{2^3}\right)$

$$f(n) = 2 + f\left(\frac{n}{2^2}\right)$$
$$= 2 + 1 + f\left(\frac{n}{2^3}\right)$$
$$= 3 + f\left(\frac{n}{2^3}\right)$$
$$= 4 + f\left(\frac{n}{2^4}\right)$$
$$\vdots$$
$$f(n) = i + f\left(\frac{n}{2^i}\right)$$

To get the initial condition $f(1)$, let $2^i = n$ — ③

$$f(n) = i + f\left(\frac{n}{n}\right)$$
$$= i + f(1) \qquad f(1) = 0 \text{ from eqn } ①$$
$$f(n) = i \quad \text{—— ④}$$

From eqn ③ $2^i = n$, take log on both sides

$$\log_2 2^i = \log_2 n$$
$$i \log_2 2 = \log_2 n \qquad \log_2 2 = 1$$
$$i = \log_2 n \quad \text{—— ⑤}$$

Substitute ⑤ in ④

$$f(n) = i$$
$$f(n) = \log_2 n$$
$$\therefore f(n) \in \Theta(\log_2 n)$$

Best and worst case remains the same for this algorithm. Hence we represent time efficiency using average case.

Eg :-

$n = 25$

$1 + Bin(n/2)$

$1 + Bin(25/2) = 12$

$1 + Bin(12/2) = 6$

$1 + Bin(6/2) = 3$

$1 + Bin(3/2) = 1$

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

$1 + Bin(1)$

25 is divided by 2 for 5 times

$\therefore$ No of digits in binary = 5

i.e 25 in binary = 11001

$\therefore$ count of digits = 5

Brute Force Approaches :-

Definition :-

- Brute force method is a straight forward method of solving a given problem based on the problem's statement and definition
- Solution is obtained without applying any strategies and logic.
- Easier to implement than a more sophisticated one because of this simplicity, sometimes it can be more efficient.

Eg :- Find GCD of 2 numbers, matrix multiplication, addition selection sort, bubble sort, sequential search etc uses brute force technique.

Sorting :-

- Process of rearranging the given elements in ascending order or descending order is called sorting.

Selection Sort :-

- As the name indicates, we first find the smallest element in the list and we exchange it with the 1st element. Obtain the 2nd smallest element in the list and exchange it with the 2nd element and so on.
- Finally, all the elements will be arranged in ascending order since, the next last element is selected and exchanged appropriately so that elements are finally sorted, this technique is called selection sort.

Eg :- consider the elements 45, 20, 40, 5, 15

Given items	Pass - 1	Pass - 2	Pass - 3	Pass - 4
A[0] = 45 ←	5	5	5	5
A[1] = 20	20 ←	15	15	15
A[2] = 40	40	40 ←	20	20
A[3] = 5 ←	45	45	45 ←	40
A[4] = 15	15 ←	20 ←	40 ←	45

- 1st smallest element is 5 and exchanged with 1st element
- 2nd smallest element is 15 and exchanged with 2nd element
- 3rd smallest element is 20 and exchanged with 3rd element
- 4th smallest element is 40 and exchanged with 4th element
- Finally all the elements are sorted in the diagram.

Design :-

The smallest element from i^{th} position onwards can be obtained using the code.

```
pos ← i                    i = 0, 1, 2, 3
for j ← i+1 to n-1
 if (a[j] < a[pos])
  pos ← j
end for
```

After finding the position of the smallest number, it should be exchanged with i^{th} position using statements.

```
temp ← a[pos]
a[pos] ← a[i]      (or)   swap (a[i], a[pos])
a[i] ← temp
```

Value of i ranges from 0 to 3
i.e, i ← 0 to n-2

Algorithm :-

```
Algorithm Selection sort (a[], n)
// sort the given elements using seledion sort
// Input : n → number of elements in array
           a → elements to be sorted are present in array.
// output : a → contains sorted array.
 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]        } (or)  Exchange(a[i], a[pos])
  a[i] ← temp
end for
```

Analysis :-

Time efficiency of this algorithm can be calculated as shown

Step-1 :- Parameter to be considered is $n \rightarrow$ input size

Step-2 :- Basic operation is comparative statement

if $(a[j] < a[pos])$

Step-3 :- Number of comparisons depends on the value of n and the number of elements, the 2 for loops are executed. Number of times basic operation is executed can be obtained as

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

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

From $\sum_{i=l}^{u} a_i \pm b_i \Rightarrow \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

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

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

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

$$= (n-1)(n-2+1) - \frac{(n-2)(n-1)}{2} \Rightarrow (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)\left[(n-1) - \frac{(n-2)}{2}\right] \Rightarrow (n-1)\left[\frac{2n-2-n+2}{2}\right]$$

$$= \frac{(n-1)\,n}{2} \Rightarrow \frac{n^2-n}{2} \approx n^2$$ By neglecting lower order terms

$f(n) = n^2 \quad \therefore f(n) \in \Theta(n^2)$

Bubble Sort :-

- Sorting algorithm that starts from the 1st element of an array and compares it with the 2nd element. If the 1st element is greater than 2nd we swap them. It continues this process unlill the end of the array, with the largest element bubbling up to the last position.
- Next pass bubbles up the 2nd largest element and so on unlill after n-1 passes, the list is sorted.

Design :-

In this technique, 2 successive items a[i] and a[i+1] are exchanged whenever a[i] > a[i+1]

Eg :- Consider the elements 40, 50, 30, 20, 10 where n = 5

	Pass-1				Pass-2			Pass-3		Pass-4
A[0] = 40	40	40	40	40	30	30	30	20	20	10
A[1] = 50	50	30	30	30	40	20	20	30	10	20
A[2] = 30	30	50	20	20	20	40	10	10	30	30
A[3] = 20	20	20	50	10	10	10	40	40	40	40
A[4] = 10	10	10	10	50	50	50	50	50	50	50
Given array	50 sinks to bottom after pass-1				40 sinks to bottom after pass-2			30 sinks to bottom after pass-3		20 sinks to bottom after pass-4

- After each pass, the larger values sink to the bottom of the array and hence called sinking sort
- At the end of each pass smaller values gradually "bubble" their way upward to the top and hence called bubble sort

Comparison that are performed in each pass are shown

Passes →	Pass-1	Pass-2	Pass-3	Pass-4
Comparing	0-1	0-1	0-1	0-1
	1-2	1-2	1-2	
	2-3	2-3		
	3-4			
	i=0 to 3	i=0 to 2	i=0 to 1	i=0 to 0
	0 to 5-2	0 to 5-3	0 to 5-4	= 0 to 5-5
	5-(1+1)	5-(2+1)	5-(3+1)	5-(4+1)
	n-(j+1)	n-(j+1)	n-(j+1)	n-(j+1)

In general,

i = 0 to n-j-1

j = 1 to n-1

Partial code can be written as

```
for j = 1 to n-1
    for i = 0 to n-j-1
        if ( A[i] > A[i+1])
        Exchange ( A[i], A[i+1])
        end if
    end for
end for
```

Algorithm :-

```
Algorithm Bubble sort (a[], n)
// Arrange the numbers in ascending order
// Input : n → number of elements in a array
           a → Array containing the elements to be sorted
// Output : a → contains sorted list in the array

for j ← 1 to n-1 do
    for i = 0 to n-j-1 do
        if (a[i] > a[i+1])
            temp ← a[i]
            a[i] ← a[i+1]
            a[i+1] ← temp
        (or) Exchange (a[i], a[i+1])
    end if
end for
end for
```

Analysis :-

Time efficiency can be obtained as shown

Step-1 :- Parameter to be considered is $n \rightarrow$ input size

Step-2 :- Basic operation is the statement

if $(a[i] > a[i+1])$

Step-3 :- Number of comparison depends on the value of n and the number of times the 2 for loops are executed. Total number of times the basic operation is executed can be obtained as

$$f(n) = \sum_{j=1}^{n-1} \sum_{i=0}^{n-j-1} 1$$

$$= \sum_{j=1}^{n-1} (n-j-1) - 0 + 1$$

$$= \sum_{j=1}^{n-1} n - j$$

$$= \sum_{j=1}^{n-1} n - \sum_{j=1}^{n-1} j$$

$$= n \sum_{j=1}^{n-1} 1 - \frac{(n-1)(n-1+1)}{2}$$

$$= n(n-1-1+1) - \frac{(n-1)n}{2}$$

$$= n(n-1) - \frac{n(n-1)}{2}$$

$$= n(n-1)\left[1 - \frac{1}{2}\right] \Rightarrow n(n-1)\frac{1}{2}$$

$$= (n^2 - n)\frac{1}{2} \approx n^2$$ By neglecting lower order terms

$$f(n) = n^2$$

$$f(n) \in \Theta(n^2)$$

By $\sum_{i=l}^{u} a_i \pm b_i = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

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

Replace n by $n-1$

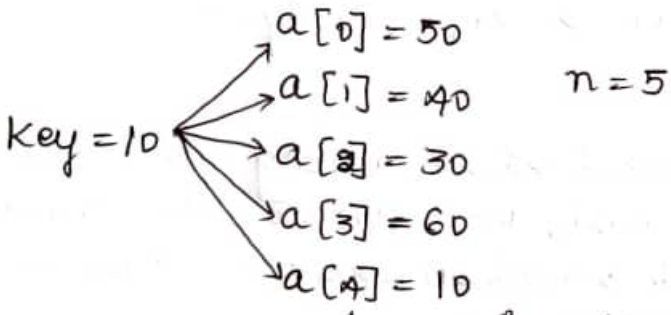
Sequential Search :-

- Also known as linear search, it is a simple searching technique In this technique, we search for a given key element in the list in linear order i.e, one after the other
- The item to be searched is often called key element

Eg :- Assume key is 10 and the list is 20, 10, 40, 25. Since key 10 is present in the list. we say search is successful. If key is 100 and after searching we say that key is not present and hence search is unsuccessful.

Design :-

Assume 10 is the item to be searched in the list 50, 40, 30, 60, 10 Observe that 10 has to be compared with a[0], a[1], a[2], a[3] and a[4] as shown.



if (Key = a[i]) return i

If key is present in the list, we return its position else we return -1 indicating search is unsuccessful.

Algorithm :-

Algorithm SeqSearch (key, a[], n)

// Algorithm searches for the key in the array 'a' which has 'n' elements and search key acts as a delimiter

// Input : n → number of elements present in the array.
a → elements in the array where searching takes place
key → Element to be searched

// output : Function returns the position if key found or else returns -1 indicating search is successful.

```
a[n] ← key
i ← 0 to n-1
while (a[i] ≠ key)
    i ← i+1
end while
if (i < n) return i
else return -1
```

Analysis :-

Basic operation is if (a[i] = key). Here the number of comparison purely depends on the position of key in the array. So, running time does not depends on value of 'n'. So, we find the worst case, best case and average case efficiences separately.

Best-case :-

- In the best case, algorithm runs fastest among all possible input. So, to analyze the best case, we have to identify the kind of input for which the basic operation count is smallest among all possible input.
- For successful search, best case occurs if the search key happens to be the first element in the lable and number of comparison required will be 1

$$\therefore f(n) \in \Omega(1)$$

Worst-case :-

- Algorithm runs for the longest duration for all possible input Here, maximum number of comparison are required.
- This case occurs if the key to be searched is present in the last position or if the key is not present in the array. So, the number of times the basic operation is executed is as shown

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

$$= n-1-0+1$$

$$f(n) = n$$

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

Average case :-

Key to be searched may be present at any position in the list from 1st to last position. So, the total number of comparison at any position is given by

$$f(n) = \frac{\text{Sum of all cases comparison}}{\text{Number of cases}}$$

$$= \frac{1+2+3+\cdots+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n(n+1)}{2n}$$

$$= \frac{n+1}{2} \approx n$$

$$f(n) = n$$

$$\therefore f(n) \in \Theta(n)$$

Brute - Force String Matching (Pattern Matching) :-

String - Matching :-

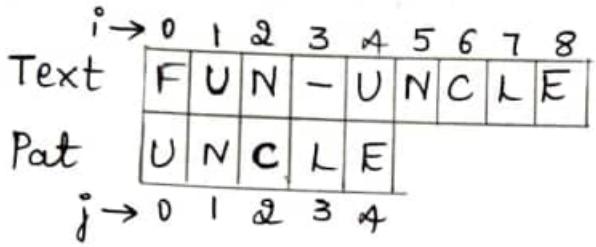
- Given a string called pattern with 'm' characters and another string called text with 'n' characters where $m \leq n$. It is required to search for the pattern in the given text string.
- If search is successful return the position of the 1st occurrence of pattern in text string otherwise return -1
- Process of searching for a pattern string in a given text string is called string matching or pattern matching.

Design :-

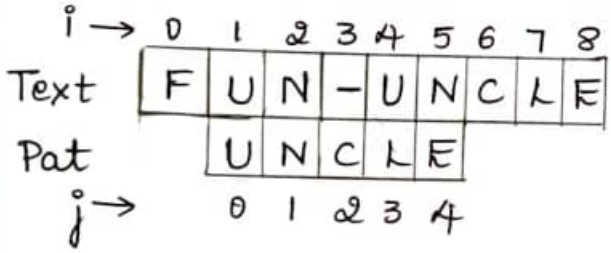
- Align the pattern string against the 1st 'm' characters of the text string and compare the characters of the pattern string with the characters of text string from left to right.

- In this process, if all characters are compared and found same, the pattern string is present in the text string and the algorithm terminates.
- If there is a mismatch, pattern string is shifted one position to the right and start comparing the pattern string with the text string.

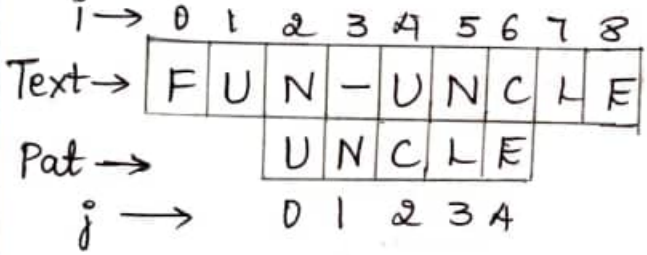
Sequence of steps to be followed while searching for a pattern string 'UNCLE' in the text string 'FUN-UNCLE' is shown below



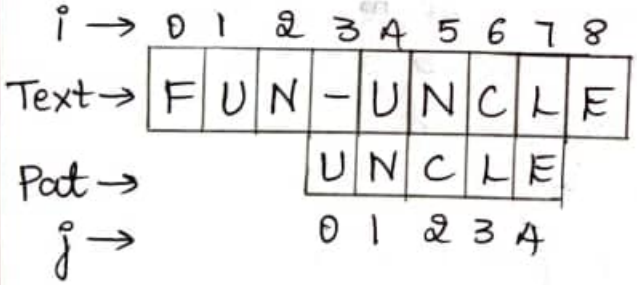
characters 'F' and 'U' are compared and they are not equal. So, slide the pattern string towards right



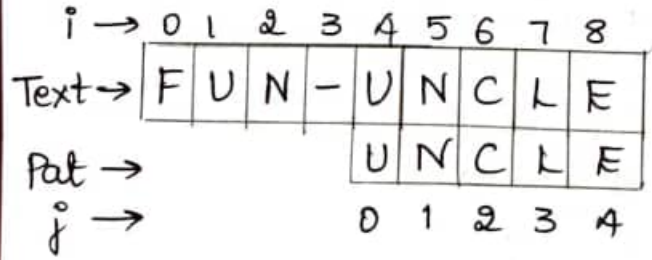
characters 'U' and 'U', 'N' and 'N' are compared and found equal. But there is a mismatch when '-' & 'C' are compared. So, slide the pattern string towards right.



Characters 'N' and 'U' are compared and they are not equal. So, slide the pattern string towards right

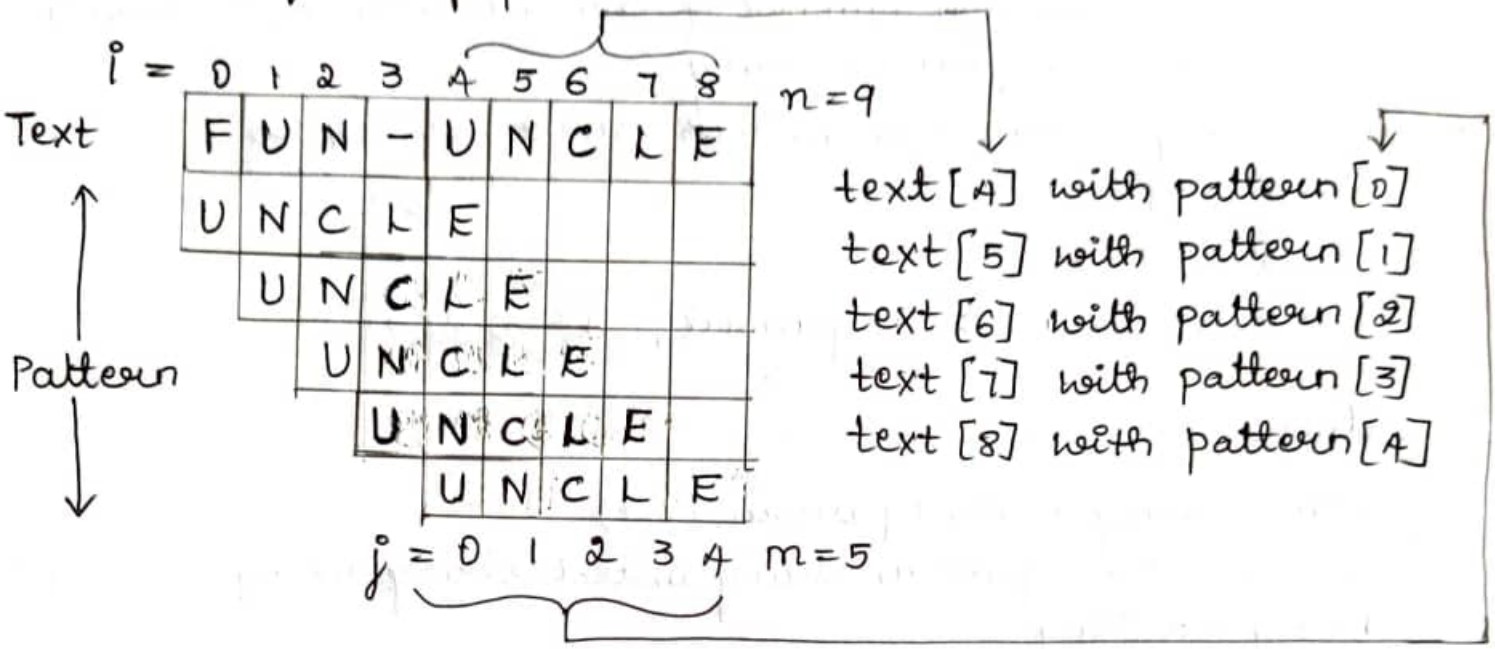


Characters '-' and 'U' are compared and they are not equal. So, slide the pattern string towards right



Pattern string found and position of 'i' has to be returned.

- From the figure, it is observed that "sliding the pattern string towards right is nothing but incrementing the index 'i' of the text string by 1".
- Entire sequence of operation can be represented as shown below



- Pattern string UNCLE being compared with text string from position 'i' whose index value is 4 can be written as

text[4] with pattern[0]
text[5] with pattern[1]
text[6] with pattern[2]
text[7] with pattern[3]
text[8] with pattern[4]

can be written as ⟹

text[4+0] with pattern[0]
text[4+1] with pattern[1]
text[4+2] with pattern[2]
text[4+3] with pattern[3]
text[4+4] with pattern[4]

text[i+j] with pattern[j]

So, keep comparing pattern[j] = text[i+j] as long as j<m as shown with initial value of j=0

```
j ← 0
while j<m and pattern[j] = text[i+j]
  j ← j+1
end while
 if j=m return i
else return -1
```

- If 'j' reaches 'm', the pattern string is found and the corresponding index 'i' in the text string can be returned.
- Index value = 4 is the last position in text string that can match a pattern string as shown in figure. Beyond this position there are not enough number of characters in text string to compare with pattern string.
- So, 'i' range from 0 to 4. This can be written as

  $i = 0$ to 4

  $= 0$ to $9-5$

  $i = 0$ to $n-m$ $\Rightarrow$ In general, $i \leftarrow 0$ to $n-m$

Algorithm :-

```
Algorithm string_match (pattern, text)
// Searches for the pattern string in text string using brute-force
   string matching
// Input : pattern[] → string of 'm' characters representing
                       pattern to be searched.
           text[] → string of 'n' characters representing a
                    text string where searching has to be carried
// Output : returns i → position of pattern in text string if found
            returns -1 → indicating pattern not found.
  n ← length (text)
  m ← length (pattern)
  for i ← 0 to n-m do
     for j ← 0 to m-1 do
       if (pattern[j] = text[i+j])
          j ← j+1
       end if
     if (j = m) return i
  end for
end for
return -1
```

Analysis :-

Time efficiency can be calculated as shown

Best - case :-

- Occurs if the pattern string to be searched is present in the beginning of the text string. If 'm' is the length of the pattern string, the number of comparison required is 'm'. Therefore, best-case time complexity is given by

$$f(n) \in \Omega(m).$$

Worst - case :-

- Occurs if the pattern is present at the end of the text or if the pattern is not present in the text.

Step-1 :- Parameter to be considered are m and n which represent the length of text and pattern string.

Step-2 :- Basic operation is pattern[j] = text[i+j]

Step-3 :- Total number of times the basic operation is executed can be obtained as

$$f(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1$$

$$= \sum_{i=0}^{n-m} m-1-0+1$$

$$= \sum_{i=0}^{n-m} m \Rightarrow m \sum_{i=0}^{n-m} 1$$

$$= m(n-m-0+1)$$

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

$$f(n) = mn - m^2 + m \approx mn$$

By neglecting lower order terms

$$f(n) = mn$$

$$\therefore f(n) \in O(mn)$$

Note :- mn is higher when compared to m^2, because n is greater than m.