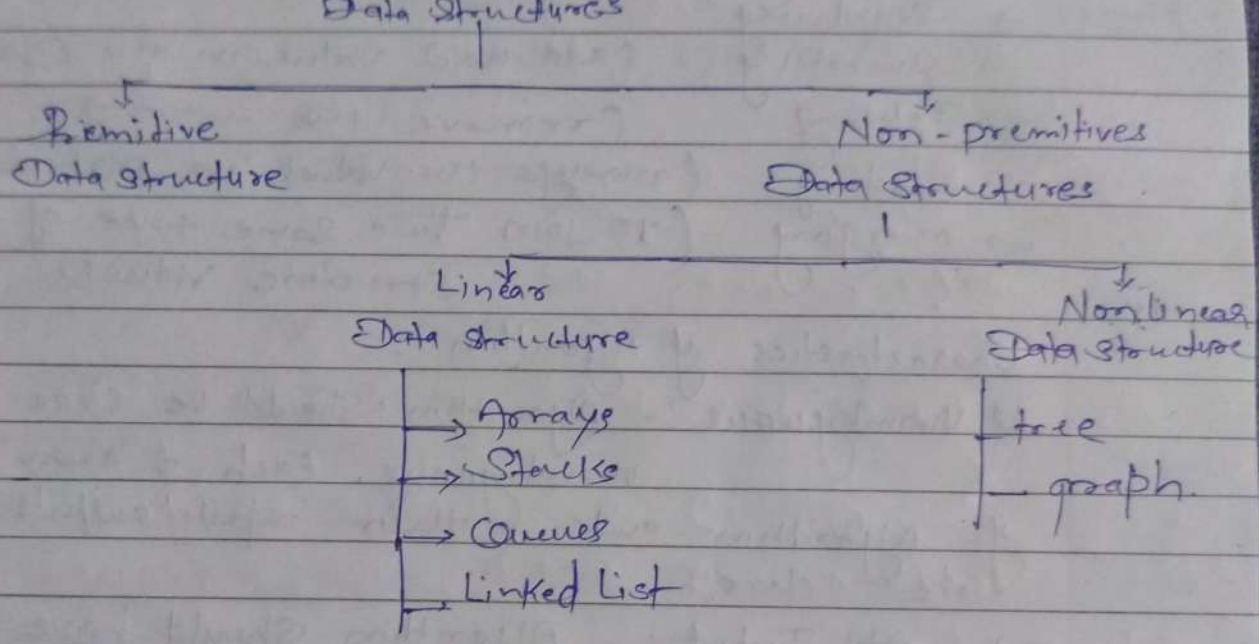
Date

UNIT - I

Data Structure :- is an arrangement of data in a Computer memory or even disk storage. An Example of several common data structures are array, linked lists, queues, stacks, binary trees and hash table.



There are two types of data structure.

1) Linear
2) Non-linear

1) Linear DS :- This Data structures involve arranging the elements in Linear fashion.

eg:- Stack, queues, Lists

2) Non-Linear DS :- elements in hierarchical order

eg. Trees, graphs.

Data Structure operations :- operations means processing the data in the data structure. the following are some important operations.

→ Traversing (visiting) (to visit each data exactly once)
→ Searching
→ Inserting (add new value in the DS)
→ Deleting (remove the —u—)
→ Sorting (arrange the values in particular order)
→ merging (To join two same type of data structure values)

Characteristics of Algorithm :-

1) Unambiguous :- Algorithm should be clear and unambiguous. Each of every steps of Algorithm and their inputs/outputs should be clear.

2) At Input :- Algorithm should have 0 or more input

3) Output :- An Algorithm should have 1 or more outputs.

4) finiteness :- Algorithm must stop after a finite number of steps.

5) Independent :- An algorithm should ~~have~~ be independent of any programming language.

Date

How to write an Algorithm :-

Problem :- Design an algorithm to add two numbers and display the result.

M1

STEP 1 : START
STEP 2 : declare three integers a, b & c
STEP 3 : define values of a & b
STEP 4 : add values of a & b
STEP 5 : store output of STEP 4 to c
STEP 6 : print c
STEP 7 : STOP

M2

STEP 1 : START ADD
STEP 2 : get values of a & b
STEP 3 : c ← a+b
STEP 4 : display c
STEP 5 : STOP

Assignment - 1

Q.1 Difference b/w primitive data structure & non-primitive DS

Frequency Count Method :-

```
Algorithm sum(A, n)
{
    s=0;                    ——— 1 unit
    for(i=0; i<n ; i++) → n+1
    {
        s= s+ A[i]; ——— n
    }
    return s;               f(n) = 2n+3
}
```

A

8	3	9	1	2
0	1	2	3	4

n=5

i=0
i=1
i=2
i=3
i=4
i=5 X

Space

A → n
n → 1
s → 1
i → 1

n+3 ⇒ O(n)

O(n)

Date

Ex

```
Algorithm Add(A, B, n)
{
for (i=0; i<n; i++) ---> n+1
  {
for (j=0; j<n; j++) ---> n*(n+1)
  {
    C[i,j] = A[i,j]+B[i,j] ---> n*n
  }}
```

Space
A → n^2
B → n^2
C → n^2
n - 1
i - 1
j - 1
S

$S(n) = 3n^2+3$

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

$O(n^2)$

Ex

```
Algorithm Multiply(A, B, n)
{
for( i=0; i<n; i++) ---> n+1
  {
for (j=0; j<n; j++) ---> n*(n+1)
  {
    C[i,j] = 0; ---> n*n
for (k=0; k<n; k++) ---> n*n*(n+1)
  {
C[i,j] = C[i,j] + A[i,k] * B[k,j]; ---> n*n*n
  }}}
```

Space
A → n^2
B → n^2
C → n^2
i → 1
j → 1
k → 1

$O(n^2)$

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

Asymptotic Notation (Allows us to analyze an algorithm running time)

i) Big-oh O – upper bound (It measures the worst case time complexity or Longest amount of time to execute the algo)

ii) omega Ω – lowerbound (– best –)

iii) theta Θ → Average (express both the lower bound & upper bound of an algorithm's running time)

1) Big-oh: The function $f(n) = O(g(n))$ iff ∃ +ve constant c and n_0

such that $f(n) \leq c * g(n)$ ∀ $n \geq n_0$

Eg $f(n) = 2n+3$

$2n+3 \leq 10 \cdot n$ ∀ $n \geq 1$

$f(n) \leq g(n)$

$\boxed{f(n) = O(n)}$

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

$f(n) \geq c \cdot g(n)$ $\quad f(n) = \Omega(n)$

Eg $f(n) = 2n+3$

$2n+3 \geq 1 \cdot n$ ∀ $n \geq 1$

3) Theta $f(n) = \Theta(g(n))$ iff ∃ +ve constant c_1, c_2 & n_0

such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Eg: $1 \cdot n \leq 2n+3 \leq 5 \cdot n$

$c_1 g(n) \quad f(n) \quad c_2 g(n)$

$\boxed{\therefore f(n) = \Theta(g(n))}$

Best, worst & Average case :- (this is Case Analysis)

1) Linear Search :

A

8	6	12	5	9	7	4	3	16	18
0	1	2	3	4	5	6	7	8	9

Key = 7 (for searching 6 comparison required)

Best case :- If you searching key element which is present at Index 0, A[0] or Beging of Index

Best Case time :- Key = 8 (If I am searching)

$B(n) = O(1)$

worst Case :- Searching a key at last Index

worst Case time. = n. (Searching key 18)

$\omega(n) = O(n)$

Average case :- $\frac{\text{all possible case time}}{\text{no. of case}}$

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

Date

Algorithm :- Algorithm is a sequence of instruction of finite steps to solve particular problem

Ex ATM (a, b)

1. Take two no's (a, b)
2. c = add(a, b)
3. printf(c);

Algorithm	Program
1) Design (time)	Implementation
2) Independent on H/w/OS	2) H/w/OS
3) Analyze	3) Testing

Characteristics of Algorithm

1. Input (0 or more)
2. output (atleast generate one output)
3. Definiteness (you can write known steps)
4. finiteness (finite no. of steps)
5. Effectiveness (don't write unnecessary statements)

How to write & analyze Algorithm

```
Algorithm swap(a, b)
{
    temp = a   → 1
    a = b;     → 1
    b = temp;  → 1
}
```

In Algorithm every single statement take one unit time

f(n) = 3 → constant time

space:
- a → 1
- b → 1
- temp → 1/3

S(n) = 3 → constant

Variable

Analyse :- 1) time (How much time it taking)
2) space (How much memory space is going to take)

1) **time Complexity**

```
for(i=0; i<n; i++)   → n+1
{
    and;             → n
}
```

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

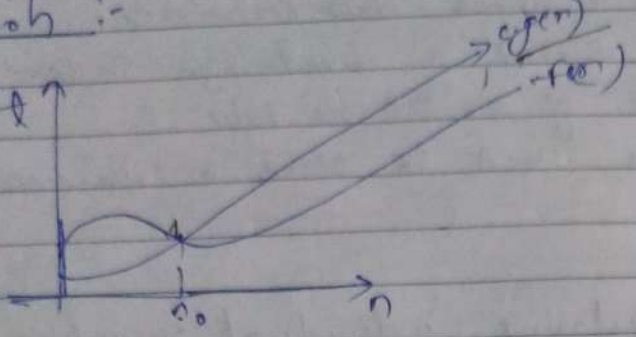
O(n) → degree of polynomial

time & space tradeoff

Date

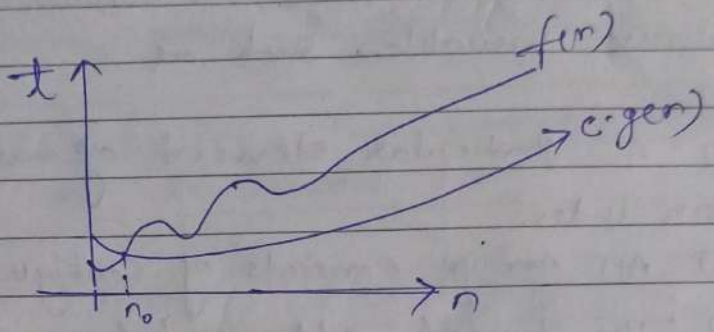
Big-oh :-

$f(n) \leq c \times g(n)$



$c > 0$
$n_0 \geq 1$
$n \geq n_0$

Big-omega (Ω) :



Eg $f(n) = 3n+2$

$g(n) = n$

$f(n) \geq c \cdot g(n)$

$3n+2 \geq c \cdot n$

↓

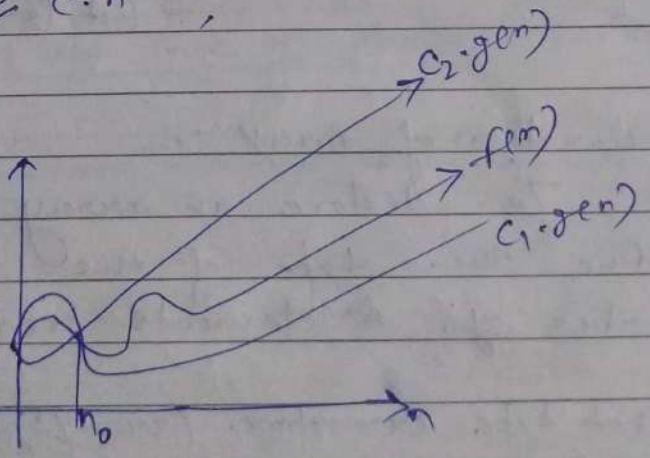
1.

Eg $f(n) = 3n+2$, $g(n) = n^2$

$3n+2 \stackrel{?}{=} \Omega(n^2)$

$3n+2 \geq c \cdot n^2$

theta (Θ) :-



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

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$; $c_1, c_2 > 0$, $n_0 \geq 1$

Eg $f(n) = 3n+2$

$g(n) = n$

check both → upper bound & lower bound

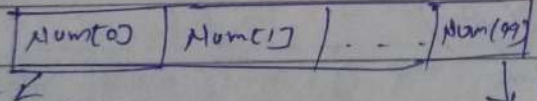
Date

Array

def: 1) Array is a linear - data structure
2) Array can stored a fixed - size sequential collection of elements of the same type.
3) An array is used to store a collection of data

Why array?
Instead of declaring individual variables, such as number0, number1, ... number99. you declare one array variable. Such as

4) A particular element of array is accessed by an index.
5) All Arrays consists of contiguous memory locations
6) The lowest address belongs to the first element and the highest address to the last element.



Declaration of Array:
1) To declare an array in C, a programmer define the type of the elements and the number of elements required by an array.

```
Data type arrayName [arraySize];
```

This is called single - dimensional array.

Eg
```
double balance[10];
int num[56];
```

Date

initializing of Array :-

int num[5] = {1, 2, 3, 4, 5};

NOTE The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

Types of Arrays :- (2-D-array)

Def: 1) A two-dimensional array is specified using row and column.

2) The C-Compiles treats a two-dimensional array as an array of one-dimensional array.

Declaration :-

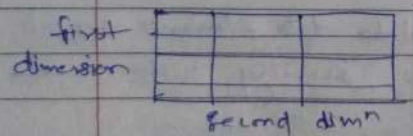
1) array must be declared before using it.

2) The declaration statements tells the compiles the name of the array,

i) data type of each element in the array

ii) and the size of each dimension

data_type array-name[row-size][column-size];



3) A two dimensional mxn array is an array that contains mxn data element and each element is accessed using i and j i<=m & j<=n

Eg: if we want to store marks obtained by three students in five different subjects, we can declare a two-dimensional array as:

int marks[3][5];

Date

m(3) → rows
n(5) → columns.

The first element of the array is denoted by
1st ele → marks[0][0] →
2nd element → marks[0][1]

Here, marks[0][0] stores the marks obtained by the first student in the first subject
marks[1][0] - store the marks obtained by the second student in the first subject

Rows / Col:	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	mark[0][3]	mark[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	[1][3]	[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	[2][3]	[2][4]

NOTE : 2D array is treated as a collection of 1D arrays. Each rows of a 2D array corresponds to a 1D array consisting of n elements, where n is the number of columns.

Address Calculation :-

1) An array stores all its ~~data~~ data elements in consecutive memory locations, storing just the base address.
2) Base address is the address of the first element in the array.
3) The address of the other element can simply be calculated using the base address.

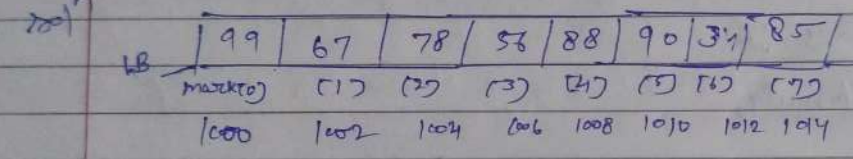
formula:

Date

Address of data element, A[k] = BA(A) + w(k - lowerbound)

Here, A is the array
k is the index of the element of which we have to calculate the address
BA is the base address of the array A,
w → is the size of one element in memory, for example size of int is 2

Q1 int marks[ ] = {99, 67, 78, 56, 88, 90, 34, 85};
Calculate the address of marks[4] if the BA = 1000



Sol Knows the storing an integers value requires 2 bytes, therefore size is 2 bytes

marks[4] = BA(A) + w(k - lower bound)
= 1000 + 2(4 - 0)
marks[4] = 1008

Calculating the Length of Array :-
The length of array is given by the number of elements stored in it.

formula :-
Length = upper bound - lower bound + 1
Upper bound → index of the last element
Lower bound → index of the first elmt

Ex Let Age[5] be an array of integers such that Age[0]=2, Age[1]=5 Age[2]=3 Age[3]=1 Age[4]=7 Show the memory representation of array and calculate the length of array.

Sol

Length = UB - LB + 1

LB = 0

UB = 4

&, Length = 4 - 0 + 1 = 5

For 2D arrays :-

Column major order formula

Address(A[i][j]) = BA + w{M(I-1) + (I-1)}

Row major order :-

Address([I][J]) = BA + w(M(I-1) + (J-1))

w → no. of bytes required to store one element

M → no. of column

N → no. of rows

I & J → subscript of the array element

Ex Consider 20×5 2D array marks which has its BA = 1000 and the size of an element = 2, Now Compute the address of the element marks[18][4] assuming that the elements are stored in row major order.

Sol Address([I][J]) = BA + w(M{(I-1) + (J-1)})

Address(marks[18][4]) = 1000 + 2{5(18-1) + (4-1)}

= 1000 + 2{5(17) + (3)}

Date

= 1000 + 2(88)
= 1000 + 176 = 1176

Applications of array:-

1) Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.
2) Many databases include one-dimensional arrays whose elements are records.
3) Arrays are also used to implement other data structures such as Strings, stacks, queues, heaps, and hash tables.
4) Arrays can be used for sorting elements in ascending or descending order.

STRING In C

In C, a string is a ~~small~~ null-terminated character array. This means that after the last character, a null character ('\0') is stored to signify the end of the character array.

Eg: Char Str[ ] = "HELLO";

Then we are declaring an array that has five characters, namely, H, E, L, L and O. Apart from these characters, a null character ('\0') is stored at the end of the string. So the internal Representation of the string becomes HELLO'\0'.

Eg:

Char Str[ ] = "HELLO";

H	E	L	L	O	\0

Beginning of String

End of String

Char ch = 'H';
Here H is a character not a string, only one memory location

H

Date

char str[ ] = "H";

H	\0

↳ Here H is a string not a character. the string H required two memory locations. one to store the character H and another to store the null character.

Eg: char str[ ] = " ";

\0

↳ Empty string.

NOTE:- C permits empty string, it does not allow an empty character.

Memory representation of character array:-

str[0]	1000	H
str[1]	1001	E
str[2]	1002	L
str[3]	1003	L
str[4]	1004	O
str[5]	1005	\0

Reading strings

char str[100];

Then str can be read by the user in three ways.

1) using scanf function
2) using gets() function
3) using getchar(), getch() or getche() function

Strings can be read using scanf() by writing

Eg: scanf("%s", str);

Date

Operations on Strings :-

1) Finding the length of string :- the number of characters in a string constitutes the length of the string.

NOTE :- Even blank spaces are counted as character in the string.

Ex LENGTH ("C. programming Is Fun!")

⇒ length : 20

Algorithm :

STEP 1 ~~SET~~ START

STEP 2 : SET I = 0

STEP 3 Repeat step 4 while STR[I] != NULL

STEP 4 SET = I+1

[END OF LOOP]

STEP 5 - SET LENGTH = I

STEP 6 : STOP

↳ Explanation :- Algorithm that calculates the length of a string. In this algorithm, I is used as an index for traversing string STR. To traverse each and every character of STR, we increment the value of I. once we encounter the null character, the control jumps out of the while loop and the length is initialized with the value of I.

Operations of String :-

Date

1) Concatenation of strings:-

↳ the library function strcat(s1, s2), which is defined in string.h Concatenates string s2 to s1

2) Comparing two strings:

s1 & s2 are two strings then comparing the two strings will give either of the following results.

1) s1 and s2 are equal
2) s1 > s2, in dictionary order, s1 will come after s2
3) s1 < s2, when in dictionary order, s1 precedes s2

The library function strcmp(s1, s2) is used to compare s1 with s2.

3) Reversing of string:-

if s1 = "HELLO", then reverse of s1 = "OLLEH". To reverse a string, we just need to swap the first character with the last, second character with the second last character, and so on.

function strrev(s1) is used to reverses all the characters in the string except the null character.

Date

Arrays of Strings :

char names[20][30];

the first index will specify how many strings are needed and the second index will specify the length of every individual string. So, here we allocate space for 20 names where each name can be a maximum 30 characters long.

Memory Representation of an array of string.

char name[5][10] = {"Ram", "Mohan", "Shayam", "Hari", "Gopal"};

name[0]	R	A	M	\0						
	M	O	H	A	N	\0				
	S	H	A	Y	M	\0	\0			
	H	A	R	I	\0					
	G	O	P	A	L	\0				

Sparse matrix :

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m×n values. If most of the element of the matrix have 0 values. then it is called Sparse matrix

Ex

$$\begin{bmatrix} 0 & 0 & 8 & 0 & 4 \\ 6 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$ — Sparse matrix

Representaing a Sparse matrix by a 2D array leads to wastage of lots of memory as zeros in the matrix are of no use in most of the cases.

triples → (Row, Column, Value).

Date

Sparse matrix Representation using Arrays :-

1) Row :- Index of row, where non-zero element is located

2) Column :- Index of column, where non-zero element is located

3) Value :- Value of the non zero element located at index → (row, column)

Column

	0	1	2	3	4
0	0	0	3	0	4
1	0	0	5	7	0
2	0	0	0	0	0
3	0	2	6	0	0

→

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

T = O(1) Constant, if is not one.

Asymptotic Notations

Let f(n) & g(n) be two positive functions

1 Big-oh Notation ⇒ (O)

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

or

f(n) is order of g(n)

iff (if only & if)

$f(n) \leq c \cdot g(n)$, $\forall n$, $n \geq n_0$

Such that There exist

& +ve Constant $c > 0$ & $n_0 > 0$

$c = \frac{1}{2}$ $n_0 = 1$

Date

Ex1

$f(n) = n+10$

$g(n) = n$

$\Downarrow$

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

$n+10 \leq c \cdot n$, $\forall\ n \geq n_0$

$\Downarrow$ $\Downarrow$

2 10

$\Downarrow$

$\boxed{n+10 = O(n)}$ proof

Ex

$f(n) = n$

$g(n) = n+10$

$\Downarrow$

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

$n \leq c \cdot (n+10)$ $\forall\ n,\ n \geq n_0$

$\Downarrow$ $\Downarrow$

1 1

also greater

$\boxed{n = O(n+10)}$

Complexity is not negative.

Date

Ex

$f(n) = n^2$

$g(n) = n$.

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

$n^2 \leq c \cdot n, \quad \forall n, \; n \geq n_0$

⇓

✗ — itself a function but c is a constant

So.

$\boxed{n^2 \neq O(n)}$ n^2 is not greater than n. By help of constant

Ex $f(n) = n - 10$

$g(n) = n$

{ if taking $n = 1, 2, 3, \ldots 10$ produce the negative function does not allow here)

⇓

$n - 10 \leq c \cdot n \quad \forall n, n \geq n_0$

⇓

1

⇓

11

$\boxed{n - 10 = O(n)}$

Date

2. Ω (omega) notation :=>

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

$f(n)$ is omega of $g(n)$

if

$f(n) \geq c\,g(n)$, $\forall n$, $n \geq n_0$

such that there exist 2 +ve. Constant

$c > 0$, & $n_0 > 0$.

Ex

$f(n) = n + 10$

$g(n) = n$

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

$n + 10 \geq c \cdot n$ $\forall n$, $n \geq n_0$.

⇓ ⇓

1 1.

proof

Ex.

$f(n) = n$

$g(n) = n + 10$

{ Constant is fraction allow but zero or negative not allow

$n = \Omega(n + 10)$

⇓

it is full

$n \geq c \cdot (n + 10)$ $\forall n$, $n \geq n_0$

⇓ ⇓

1/10 10

Date

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

Prof this $n = \Omega(n^2)$

$\Downarrow$

$n \geq c \cdot (n^2)$

$\Downarrow$

$\not{n}$ — not allowed, because it is function.

$$\boxed{n \neq \Omega(n^2)}$$

3. Theta - Notation (Θ) :⇒

$$\boxed{f(n) = \Theta(g(n))}$$

iff.

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

&

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

Ex. $f(n) = n + 10$
$g(n) = n$

$c_1 = 2$, $c_2 = 1$ → $n_0 = 10$

$\Downarrow$

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

Date

Satisfy 1. $n+10 \leq c \cdot n$, $\forall n$, $n \geq n_0$

$\Downarrow$

2

$\Downarrow$

10

$n_0 = 10$

Satisfy 2. $n+10 \geq c \cdot n$, $\forall n$, $n \geq n_0$

$\Downarrow$

1

$\Downarrow$

1

Both are possible

Ex 2

$f(n) = n$

$g(n) = n$

$c_1 = 1$ $n_0 = 1$

$c_2 = 1$

$\Rightarrow f(n) = \Theta(g(n))$

Big-oh possible ✓ : $n \leq c \cdot n$, $\forall n$, $n \geq n_0$

$\Downarrow$

1

$\Downarrow$

1

$n_0 = 1$

Omega possible ✓ : $n \geq c \cdot n$, $\forall n$, $n \geq n_0$

$\Downarrow$

1

$\Downarrow$

1

Ex

$f(n) = n$. Mathematical ✓ Asymptotical ✗

$g(n) = n^2$

(Ans) $f(n) = o(g(n))$

$\Downarrow$

Date

1. $n \leq c \cdot n^2$ ✓ n, $n \geq n_0$

possible

$\Downarrow$

1

$\Downarrow$

1

$\Downarrow$

$+\infty$

2. $n \geq c \cdot n^2$ — not possible

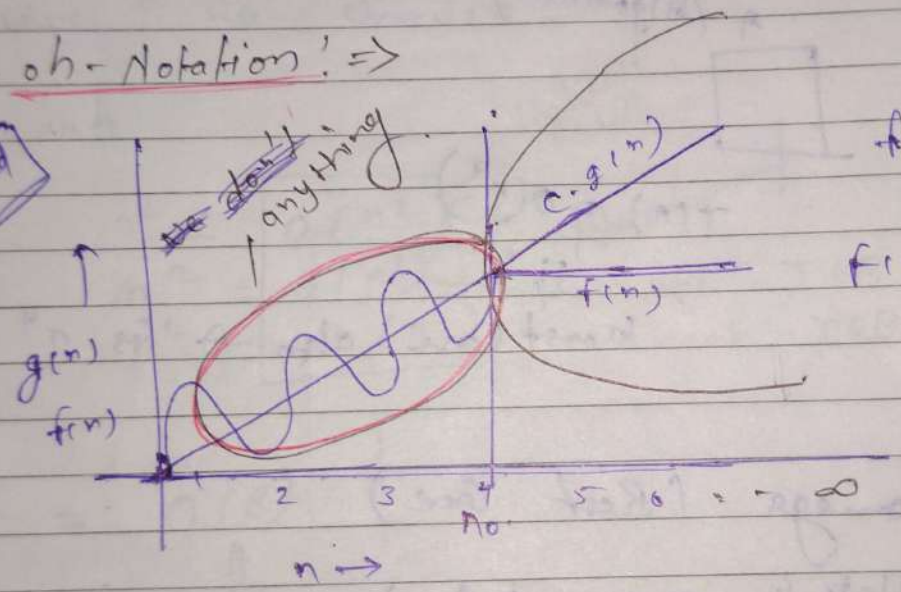
$\Downarrow$

$\frac{1}{n}$

$n \neq \Theta(n^2)$

Big-oh-Notation :=>

upper Bound



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

$\Downarrow$

$f(n) \leq c \cdot g(n)$

$\forall n,\ n \geq n_0$

Omega notation

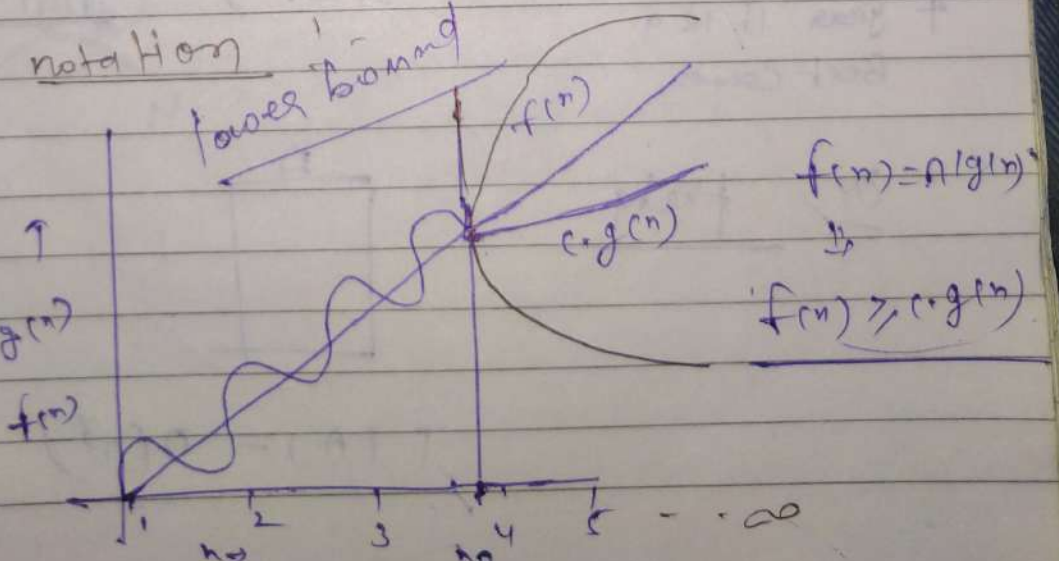
lower bound

$T(A) = \Omega(n^4)$

$\Downarrow$

Best case of A also

A is n^3

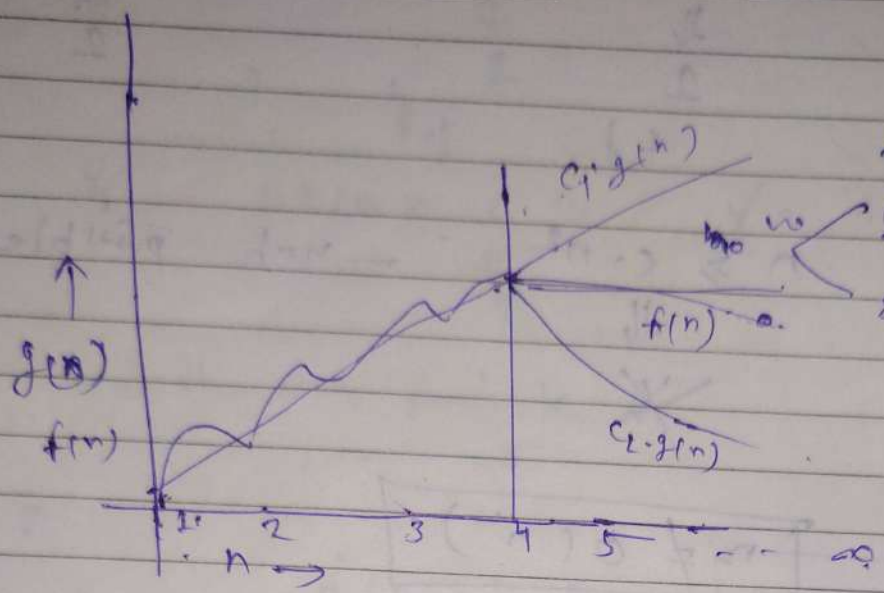


$f(n) = \Omega(g(n))$

$\Downarrow$

$f(n) \geq c \cdot g(n)$

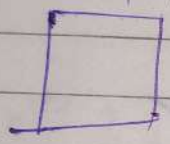
3. Theta - Notation (Θ)



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

1. $f(n) \leq c_1 g(n)$ ↓↑
2. $f(n) \geq c_2 g(n)$

A (Algorithm)



$T(A) = \Theta(n^4)$

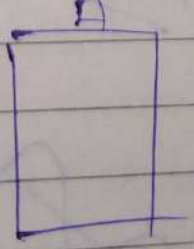
||

Worst case also A is n^4

Ex omega (Best Case)

Btech complete in 4 years it is a Best case $T(4 \cdot Btech) = \underline{\Omega(4)}$

Ex theta



$T(A) = \Theta(n^3)$

Big-oh notation purpose → find upper bound tight / not tight

Date

4f

$T(n) = \Omega(n^3)$. Best case

$\Downarrow$

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

Time Complexity in B.C and W.C are same indicate the theta

⇒ if upper bound and lower bound are same, then only theta notation is possible.
Otherwise no chance.

Big-oh

tightest UB

$n^2 =$ $O(n^2)$ ← upper bound

$n^2 =$ $O(n^3)$ → not TUB

$n^2 =$ $O(n^5)$ → not TUB

$A = O(B)$

$\Downarrow$

UB

T N

Stack

Date

Searching :- Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. if value is not present in the array, the searching process displays message and in this case searching is said to be unsuccessful.

There are two popular methods for searching the array elements:

i) linear search
ii) Binary search.

The algorithm that should be used depends entirely on how the values are organized in the array.

for ex. if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient in terms of Complexity

1) **Linear Search :-** Linear search, also called Sequential search is a very simple method used for searching an array for a particular array. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

Ex. i/p A [10 20 30 1 2 3 11 21 31]
1 2 3 4 5 6 7 8 9

x = 3 | x = 31 | x = 56

STEP5: IF POS = -1
PRINT "Value is not present in array"
[END IF]
STEP6: STOP

```
         arr  no. of element
          ↓    ↓   → key
Linear (a, n, x)
{
  for (i=1; i≤n; i++)
  {
    if (a[i] == x)
      return (i)
  }
  return (-1)
}
```

```
Algo
LINEAR SEARCH (A, N, x)
Step1: SET POS = -1
STEP2: SET I = 1
STEP3: Repeat step4 while I<N
STEP4: IF A[I] = x
         SET POS = I
         PRINT POS
         Go TO STEP 6
       [END OF IF]
       SET I = I+1 (END loop)
```

Complexity of Linear Search Algorithm :- Linear Search executes in O(n) time when n is the number of elements in the array. the best case of Linear Search is when x(Key) is equal to the first element of the array. in this case only one comparision will be made. The worst case will happen when either x(Key) is not present in the array or it is equal to the last element of the array. in both the cases. n Comparisions will have to be made.

The performance of the linear Search algorithm Can be improved by using a sorted array.

2. BINARY SEARCH :- Binary search is a searching algorithm that works efficiently with a sorted list [sorted array].

Ex int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
and the value to be searched is val = 9. the algorithm execute in the following manner.

BEG = 0, END = 10, MID = (0+10)/2 = 5

Now, val = 9 and A[MID] = A[5] = 5

Date

A[5] is less than VAL, therefore, we now search for the value in second half of the array, so we change the values of BEG and MID

Now, BEG = MID+1 = 6, END = 10, MID = $\frac{(6+10)}{2}$ = 8

val = 9 and A[MID] = 8.

A[8] is less than val, therefore we now search for the value in the second half of the segment. So again we change the values of BEG and MID

Now,

BEG = MID+1 = 9. END = 10, MID = $\lfloor \frac{(9+10)}{2} \rfloor$ = 9

val = 9 and A[MID] = 9.

Here,

BEG = Beginning

END = ending

MID = (BEG + END) / 2

~~BEG~~ = i, j

i/p: [10 | 20 | 30 | 40 | 50 | 60 | 70]

1 2 3 4 5 6 7

BinarySearch (a, i, j, x)

{ int mid;

if (i == j) → 2 comparison → take constant time O(1)

{

if (a[i] == x)

return (i);

else return (-1); }

else { mid = (i+j)/2

if (a[mid] == x) return (mid)

else

if i → position

x → element

```
if (a[mid] > x)
    BS(a, i, mid-1, x)        → T(n/2)
else
    BS(a, mid+1, j, x)        → T(n/2)
}
```

Complexity of Binary Search Algorithm :-

The Complexity of the binary search algorithm can be expressed as $f(n)$ where n is the number of elements in the array. The Complexity of the algorithm is calculated depending on the number of comparisons that are made. In the Binary Search algorithm, we see that with each comparisons the size of the ~~element~~ segment where search has to be made is reduced to half. Thus we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as

$$f(n) = \log(n)$$

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + c & \text{if } n > 1 \end{cases}$$

c is Constant

$$T(n) = T(n/2) + c$$
$$= T(n/2^2) + c + c$$
$$= T(n/2^3) + c + c + c$$
$$= T(n/2^k) + kc$$
$$= T(2^k/2^k) + c \cdot \log n$$

$n/2^k = 1$
$n = 2^k$
$k = \log_2 n$

Date

$= T(1) + c \cdot \log_2 n$

$= 1 + c \cdot \log_2 n \rightarrow O(\log_2 n)$