Date :
Page No.

DAY 3 :
Session 3 & ~~lesson~~ .
Session 4

* Data structure :-

↳ A data structure is a storage that is used to store & organise data. It is a way of arranging data on a computer so that it can be accessed & updated efficiency.

↳ Data structure is a specilized format for organising, processing, retrieving & storing data.

⇒ Every application, software or programs foundation consists of two components :- "data" is information & "algorithms" are instructions that turn data into something useful to programming.
In simple way :

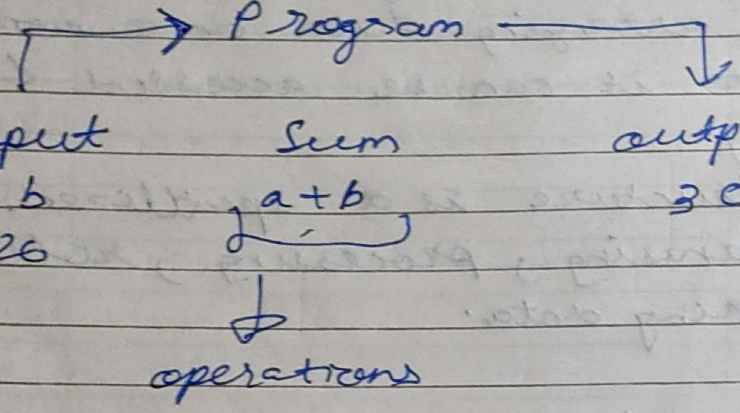
Related Data + operation on the data
= Data structure.

Data structure + Algorithms = Programs.

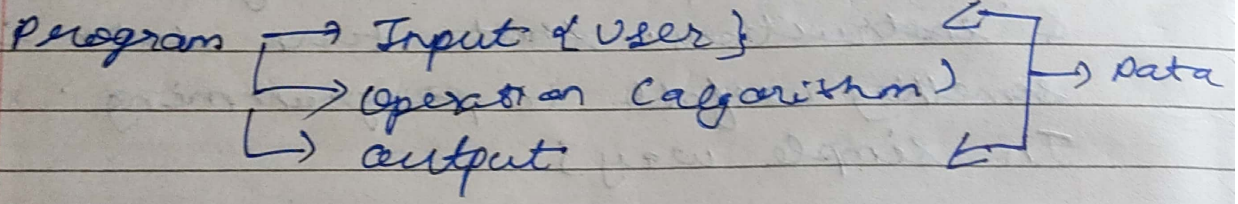
Date :
Page No.

* Program :-

↳ A computer program is a set of instructions written in a programming written in a programming language that a computer follows to execute a specific task.



Basically,



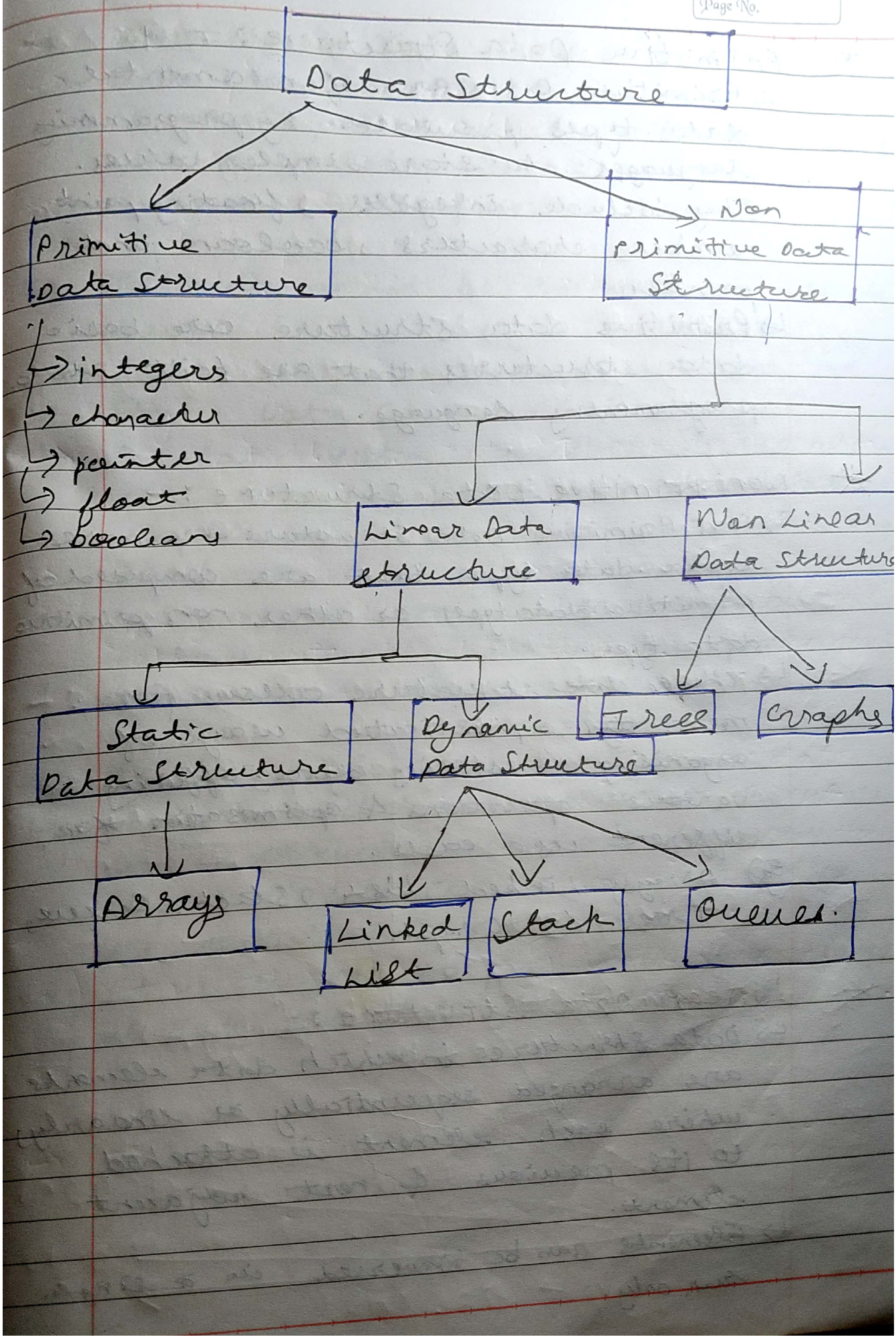
Program → Operations ⇒ algorithm
→ Data ⇒ Data structure

Therefore,

⇒ For writing an effident program, a person should known D.S.A.

Date :
Page No.

* Data type	Data Structure.
↳ Datatype is the form of a variable to which a value can be assigned. It defined that the particular variable will assign the values of the given data type only.	↳ Data structure is a collection of different kinds of data. That entire data can be represented using an object & can be used throughout the program.
↳ It can hold value but not data. Therefore it is dataless.	↳ It can hold multiple types of data within a single object.
↳ In the case data type, the value of data is not stored because it only represents the type of data that can be stored.	↳ While in the case of data structures, the data & its value acquire the space in the computer's main memory. Also data can hold different kinds & types of data within one single object.
Eg) int, float, double.	Eg) Stack, Queue, Tree.
↳ There is not time complexity	↳ Time complexity varies & hence important to consider.


Data Structure
Primitive Data Structure
integers
character
pointer
float
boolean
Non primitive Data Structure
Linear Data structure
Non Linear Data Structure
Static Data Structure
Dynamic Data Structure
Trees
Graphs
Arrays
Linked List
Stack
Queues

Date :
Page No.

* Primitive Data Structure :—
  ↳ Primitive D.S. are fundamental data types provided by programming languages to store simpler values. They include integers, floating point number, characters, booleans and pointers
  ↳ Primitive data structure are basic data structures that are built into a programming language.

* Non Primitive Data Structure :—
  ↳ Non Primitive Data Structure are more complex data types that are composed of primitive datatypes or other non primitive data types.
  ↳ These data structure allow for more sophisticated ways of organising & storing data, offering various operations & optimisation for different use cases.
  Eg) Arrays, Linked List, Stack, Queue, Trees & Graph

Linear Data Structure :
↳ Data Structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous & next adjacent element.
↳ Elements can be traversed in a single run only.

* Non Linear Data Structure :-
↳ Data Structure where data elements are not placed sequentially or linearly.
↳ Data element attached in sequential manner.

↳ Element cannot be traverse in a single run.

* Static Data structure :-
↳ A static data structure is one whose size and structure are fixed at compile time & cannot be changed during the execution of a program.
↳ The content can be modified but without changing the memory space allocated to it.

* Dynamic Data Structure :-
↳ A dynamic data structure is one whose size & structure are not fixed at compile time & can be changed during the execution of a program.
⇒ It is designed to ~~facil~~ facilitate change of data structure during runtime.

Date :
Page No.

Session 3 & ~~Session 2~~ Session 4

* Time Complexity :-

↳ The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

⇒ Generally, runtime of an algorithm depends on :

1) whether it is running on single processor machine or multi processor machine.
2) whether it is a 32 bit machine or 64 bit machine.
3) Read & write speed of the machine.
4) Input Data
5) The time taken to perform Arithmetic operations, logical operations, return value etc

* Calculating Time Complexity :-

```
int fun ( int a, int b ) {
        return a + b ;
}
```

Time Required :-

To calculate a+b ⇒ 1 unit of time

For returning a+b ⇒ 1 unit of time.

Total Time ⇒ 2 unit of time

{ Hypothetical }

⇒ Consider two developers who created an algorithm to sort n numbers. A & B did this independently.

↳ when ran for input size n, following result were recorded.

input size (n)	A's algo	B's algo
10	90 ms	122 ms
200	110 ms	124 ms
300	180 ms	131 ms
4000	2 s	890 ms

We can see that initially A's algorithm was better for smaller input but as the input size increases B's algo was better.

Q) Who's algo is best ?

=

Date :
Page No.

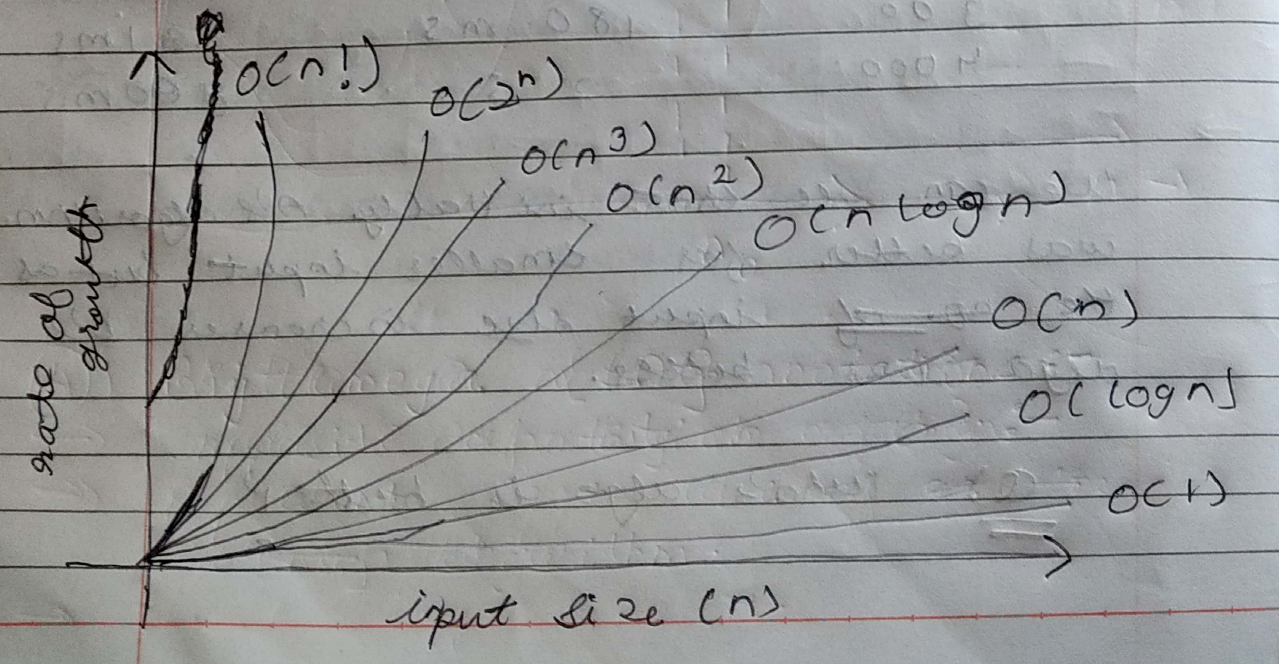
⇒ The time required by an algorithm falls under three types :-

1) BEST Case :-
• minimum time required for program execution.

2) AVERAGE Case :-
• average time required for program execution.

3) Worst case :-
• maximum time required for program execution.

* Order of Growth of an Algo :-
↳ Order of growth defines the amount of time taken by any program with respect to the size of the input.



Date :
Page No.

O(1) ⇒ BEST
O(log n) ⇒ GOOR
O(n) ⇒ FAIR
O(n log n) ⇒ BAD
O(n²), O(n³), O(2ⁿ), O(n!) ⇒ WORST

⇒ O(1) ⇒ Constant Time complexity
⇒ O(log n) ⇒ Logarithmetic Time complexity
⇒ O(n) ⇒ Linear Time complexity
⇒ O(n log n) ⇒ Linearithmic Time complexity
⇒ O(n^c) ⇒ ~~Quadratic~~ Polynomial Time complexity.
⇒ O(c^n) ⇒ Exponential Time complexity.
⇒ O(n!) ⇒ Factorial Time complexity

* Asymptotic notations :- ~~mathematical notations~~
↳ Asymptotic notations are, used to represent the complexity of an algorithm. With the help of ~~an algorithm~~ these notations we can analyse runtime performance of an algorithm by defining a time function f(n) where n is a input size. is dependent of machine & programming style.

* Asymptotic analysis :-
↳ The study of change in performance of the algorithm with the change is the order of the input size is defined as asymptotic ~~notations~~ analysis.

Date :
Page No.

* Types of Asymptotic notations :-

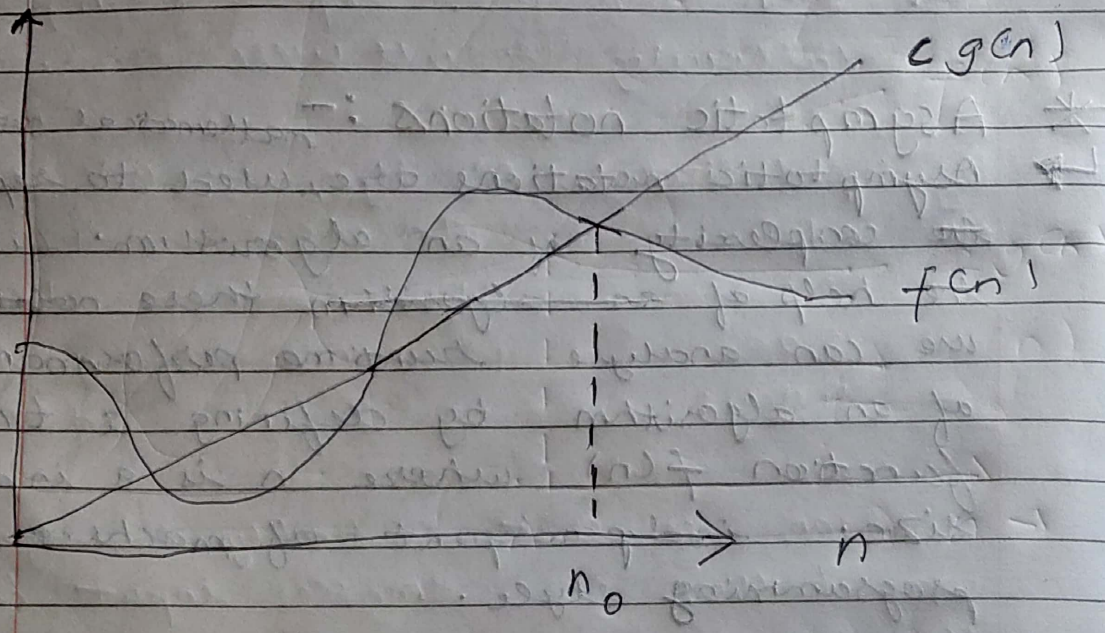
(a) Big oh (O) notations :-

↳ Represent upper bound of the running algorithm.

↳ Represent worst case of an algorithm.

⇒ Mathematically,
If $f(n)$ describes running time of an algorithm, then $f(n) = O(g(n))$ if there exist positive constant c & n_0 such that

$$0 \leq f(n) \leq c\,g(n) \quad \forall \quad n \geq n_0$$



(b) Big omega $\{\Omega\}$ notations :-

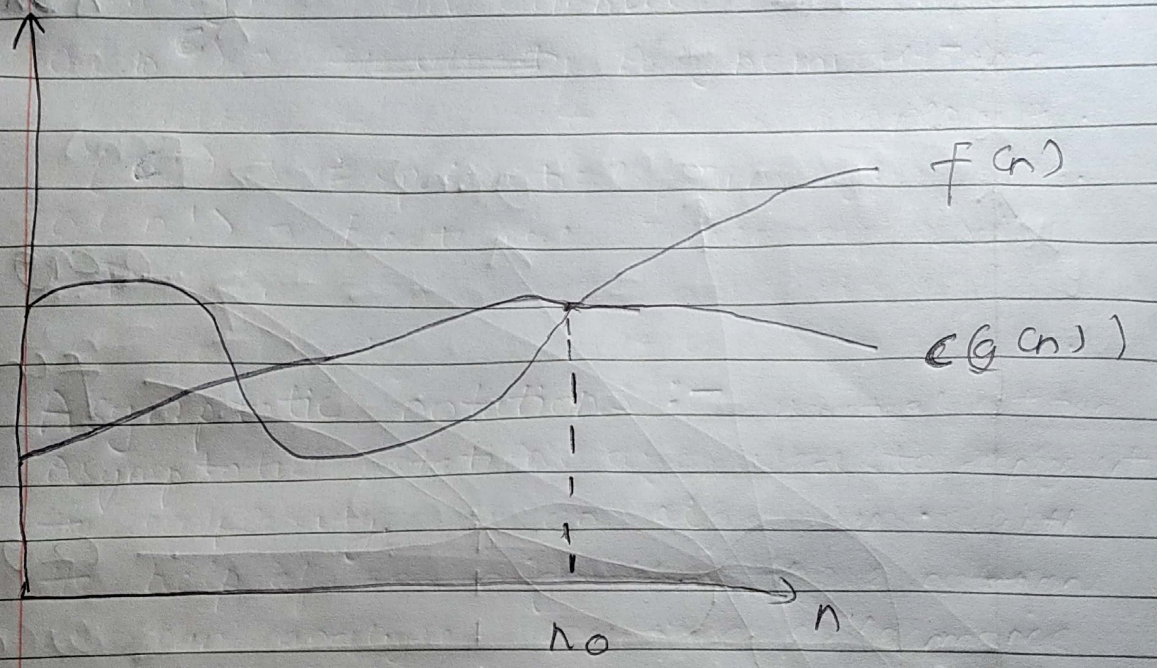
→ opposite of Big oh notations

→ Represent Best case of the algorithm.

→ Represent lower bound of the running algorithm.

→ Mathematically,
If $f(n)$ describes the running time of an algorithm then $f(n) = \Omega(g(n))$ if there exist positive constant c & n_0 such that

$$0 \leq c\,g(n) \leq f(n) \quad \forall\ n > n_0$$



(c) Big Theta (Θ) notations :-

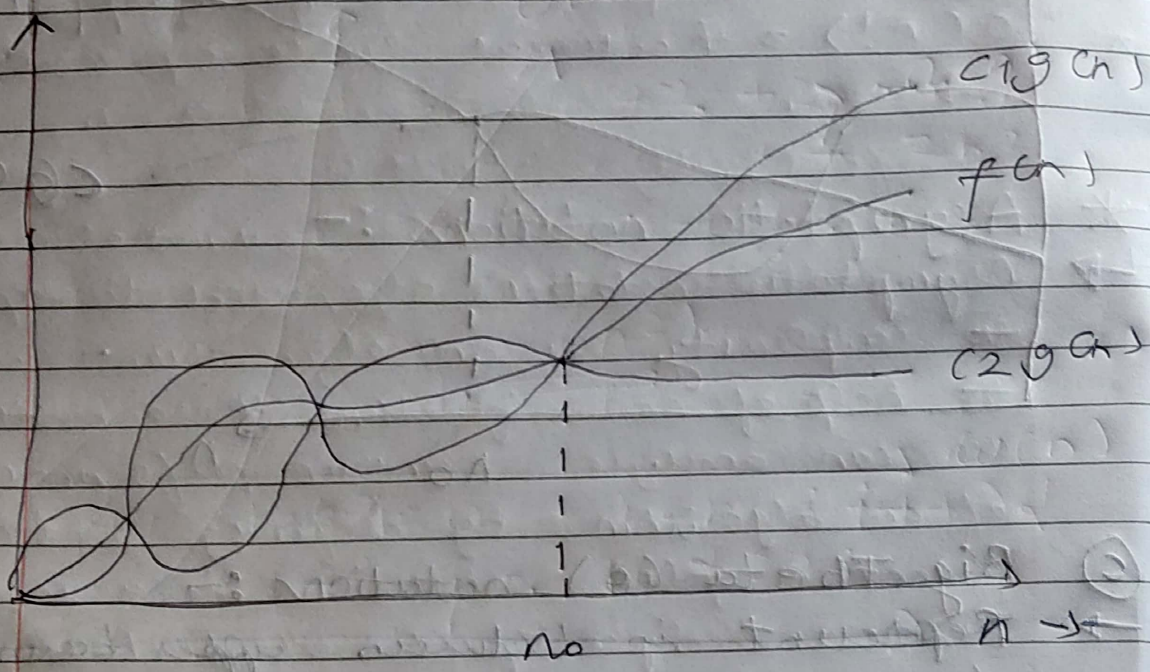
→ Represent is between upper bound & lower bound of an algorithm
↳ It is use when worst case & best case is same.

→ Represent average case complexity of an algorithm.

→ Mathematically,

↳ If $f(n)$ describes the running time of an algorithm then $f(n) = \Theta(g(n))$ if there exist positive constant c_1 & c_2 & n_0 such that:

$$0 \leq f(n) \leq c_1 g(n) \quad , n > n_0$$
$$0 \leq c_2 g(n) \leq f(n) \quad , n > n_0$$
$$\text{i.e. } 0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \quad ; \quad n > n_0$$



4) Little O notation or Small o notation {o} :-

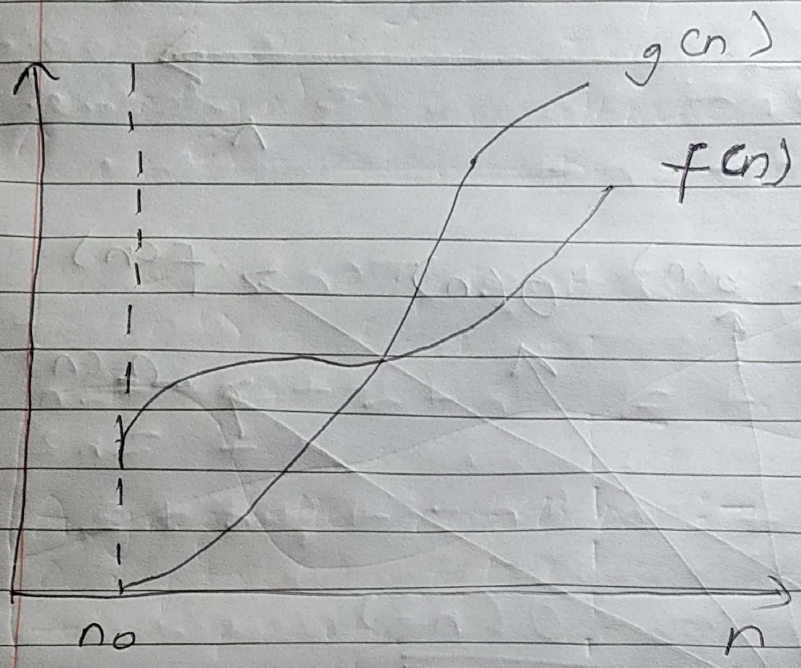
→ Represent upper bound that is not tight.

→ Loose upper bound.

→ Mathematically;

↳ If $f(n)$ describes running time of the algo, then $f(n) = O(g(n))$ for every positive constant C, there exist an integer constant for every such that positive constant C such that $0 \leq f(n) \leq C g(n)$.

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



5) Little omega notations or Small omega notations (ω)

→ Represent lower bound that is not tight.

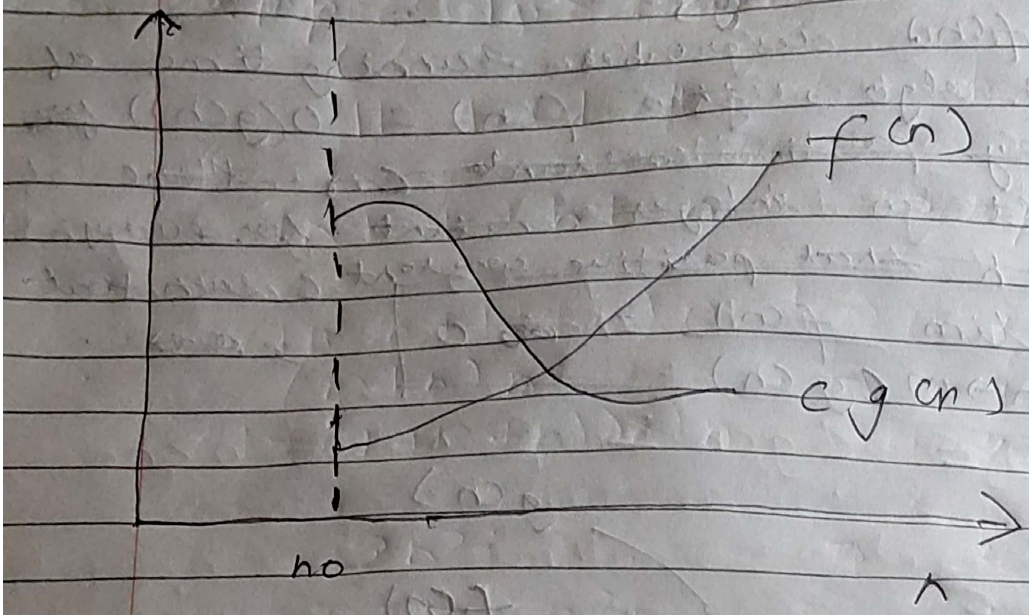
→ Loose lower bound.

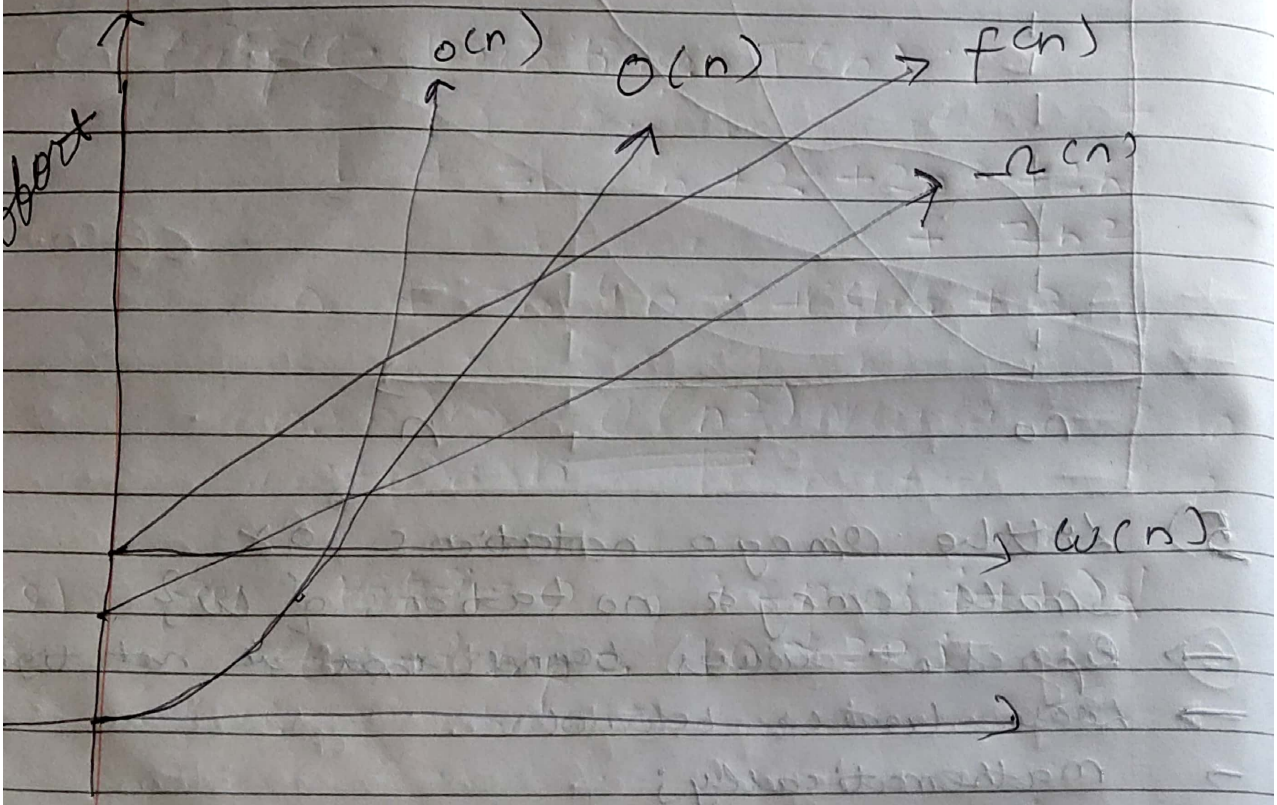
→ mathematically;

↳ If $f(n)$ describes the running time of algorithm, then $f(n) = \omega(g(n))$ for every positive constant C. C such that.

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

Date :
Page No.





Date :
Page No.

* Questions Based on Time complexity.

a) for (~~int~~ i=0 ; i<n ; i++) {
    for (~~int~~ j=0 ; j<n ; j++)
    {
        temp = 10 * 30 ;
    }
}

Ans for (i=0 ; i<n ; j++)

1 + 1 + 1 ⇒ Cost

1 + n+1 + n ⇒ Repetitions.

+ 2n + 2 ⇒ Total.

for (j=0 ; j<n ; j++)

1 1 1 ⇒ Cost

1 + (n+1) + n ⇒ Repetitions

(1 + (n+1) + n) n ⇒ Nested for loop.

2n2 + 2n ⇒ Total.

Date :
Page No.

temp = 10 * 30

1 + 1 ⇒ Cost

n + n ⇒ repetitions.

(n+n) n ⇒ nested for loop

2n² ⇒ total.

Total runtime T(n).

= 2n + 2 + 2n² + 2n

+ 2n2

= 4n² + 4n + 2

⇒ O(n²)

2) for (i=0 ; i<n ; i++) {

temp = 10 * 30 ;

}

⇒ for (i=0 ; i<n ; i++) {

1 + 1 + 1 ⇒ cost

1 + n+1 + n ⇒ Repetitions

2n + 2 ⇒ Total

Date :
Page No.

temp = 10 * 30

1 + 1 ⇒ cost

n + n ⇒ repetition

n^2 ⇒ Total.

Total runtime $T(n) = 2n + 2 + n^2$

$= n^2$

Q) for (i = 0 ; i < n ; i++) {
  for (j = 0 ; j < n ; j++) {
    for (k = 0 ; k < n ; k++) {
      temp = 10 * 30 ;
    }
  }
}

Ans) for (i = 0 ; i < n ; i++)

1 + 1 + 1 ⇒ cost

1 + n+1 + n ⇒ Repetitions

2n + 2 ⇒ Total

for (j = 0 ; j < n ; j++)

1 + 1 + 1 ⇒ cost

1 + (n+1) + n ⇒ Repetitions

n (1 + (n+1) + n) ⇒ Nested loop

$2n^2 + 2n$ ⇒ Total.

Date :
Page No.

for (k=0 ; k<=n ; k++)

1 + 1 + 1 ⇒ Cost

1 + n+1 + n ⇒ Repitition

(1+(n+1)+n) n ⇒ Nested for loop

((1+(n+1)+n))n) n ⇒ Nested for loop again

$2n^3 + 2n^2$ ⇒ Total

temp = 10 * 30

1 + 1 + 1 ⇒ Cost

n + n ⇒ Repitition

(n+n)n ⇒ Nested for loop

((n+n)n)n ⇒ Nested for loop again

$2n^3$ ⇒ Total.

Total runtime ⇒

$2n+2 + 2n^2+2n + 2n^3+2n^2 + 2n^3$

$4n^3 + 4n^2 + 4n + 2$

n^3

Q

```
p = 0
for (i = 0; i * i <= n; i++)
{
    statement;
    p = p + i * i;
}
```

A

i	p	p + i
1	0	0 + 1 ⇒ 1
2	1	1 + 2 ⇒ 3
3	3	3 + 3 ⇒ 6
4	6	6 + 4 = 10
K		0 + 1 + 2 + 3 + ... + K

$p > n \Rightarrow$ exact for loop

$p = \frac{K(K+1)}{2} \Rightarrow \underline{K^2}$

$p > n$

$K^2 > n$

$K > \sqrt{n}$

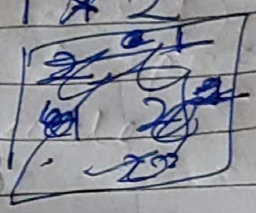
$\sqrt{n}$

Q) for(i = 1; i < n; i = i * 2)
{
    statement;
}

Ans

i
1
2
4
.
.
.
K

i * 2



2^1
2^2
2^3

2^K

$i \geq n$ ⇒ Breaking statement
$i \geq K$
$i \geq 2^K$

$2^K \geq n$
$K \log_2 2 \geq \log_2 n$

$K = \boxed{\log_2 n}$

* Result :-

{ i++ / i-- } $i <= n$ ⇒ n

{ (i = i * K) / i = i / K } $i <= n$ ⇒ $\log_2 n$

{ i = i + p } $p < n$ ⇒ $\sqrt{n}$

Q)
```
for (i=0; i<n ; i++) {
    for (j=1; j<n ; j*=2)
    {
        statement;
    }
}
```
n * log n

Ans
```
for (i=0; i<n ; i++)
```
↳ n+1 ⇒ (n)

```
for (j=1; j<n ; j*2)
```

j	j*2
1	1*2 ⇒ 2
2	2*2 ⇒ 2^2 ⇒ 4
4	4*2 ⇒ 2^3 ⇒ 8
⋮	⋮
K	K*2 ⇒ 2^K

$j < n$
$j < K$
$j \leq 2^k$
$j > K$ ⇒ Breaking statement
$2^K > n$

$K = \log_2 n$

$T(n) \geq n \log n + n + n \log n$

$\boxed{T(n) \geq n \log n}$

Q What is time & space complexity of following code:

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

assume rand() is O(1) of time & space:

Ans

```
int a = 0, b = 0;                    ⇒ 2
for (i = 0; i < N; i++)
```

1 n+1 n ⇒ 2n+2

a = a + rand() ⇒ 1 + 1 = 2

Date :
Page No.

for (j=0 ; j<=m ; j++)

1 m+1 m

2m + 2

b = b + rand() ⇒ 1+1 ⇒ 2

T(n) = 2 + 2n + 2 + 2 + 2m + 2

⇒ 2n + 2m.

⇒ n + m

O(n+m)

⇒ space T.C.

T(n) = 2 + 2

⇒ 4

O(1)

Q What is time & space complexity

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a = a + j;
    }
}
for (k = 0; k < N; k++) {
    b = b + k;
}
```

Ans) int a = 0, b = 0; ⇒ 2

for (i = 0; i < N; i++) ⇒ 2n + 2

for (j = 0; j < N; j++) ⇒ (2n + 2)n

a = a + j ⇒ 2n²

for (k = 0; k < N; k++) ⇒ 2n + 2

b = b + k ⇒ 2n

O(n²)

Date :
Page No.

space

$T(n) = 2 + 2$

$= 4$

$= \boxed{O(1)}$

Q what is time compexcity of this code :-

```
int a=0;
for(i=0; i<N; i++){
    for(j=N; j>i; j--){
        a = a+i+j;
    }
}
```

A

```
int a=0;                          → (1)
for(i=0; i<N; i++)                → 2n+2
                                    ↳ n
for(j=N; j>i; j--)                → (N+0+N)N
                                    (2N)N
                                    ↳ (n)n
                                    ↳ n^2
```

n^2

Date :
Page No.

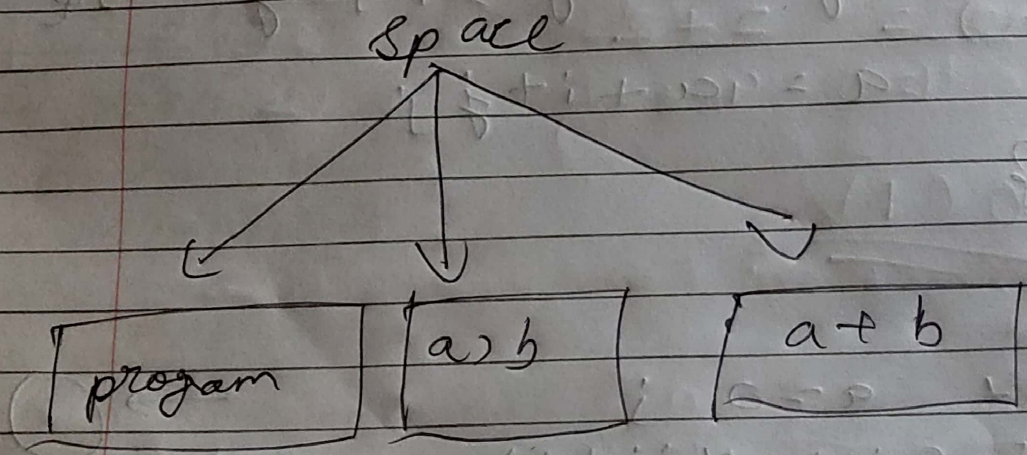
* Space Complexity :-

⇒ For any algo, memory required to

1) Instruction Space :-
↳ Store compiled version of program instruction

2) Data space :-
↳ Store all variables & constants.

3) Environmental Stack :-
↳ store information of partially executed functions at the time of function calls.



Instruction space | Data space | stack space

Space complexity is the amount of memory space of an algo requires to execute relative to the size of the input data.

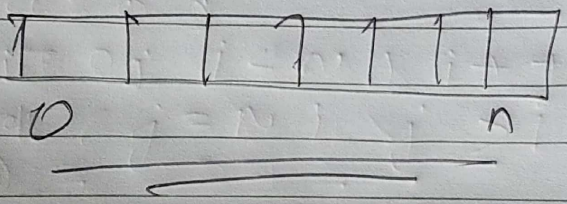
character	⇒	1 bytes
integer	⇒	2 bytes
floating point	⇒	4 bytes
double	⇒	6 or 8 bytes

Q

```
int sum ( int A[] , int n ) {
int sum = 0; i;
for (i=0; i<n; i++)
sum = sum + A[i];
return;
}
```

A

int A[]



$2 * n$.

$T(n) = 2n + 2 + 2 + 2 \Rightarrow 2n + 6$

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