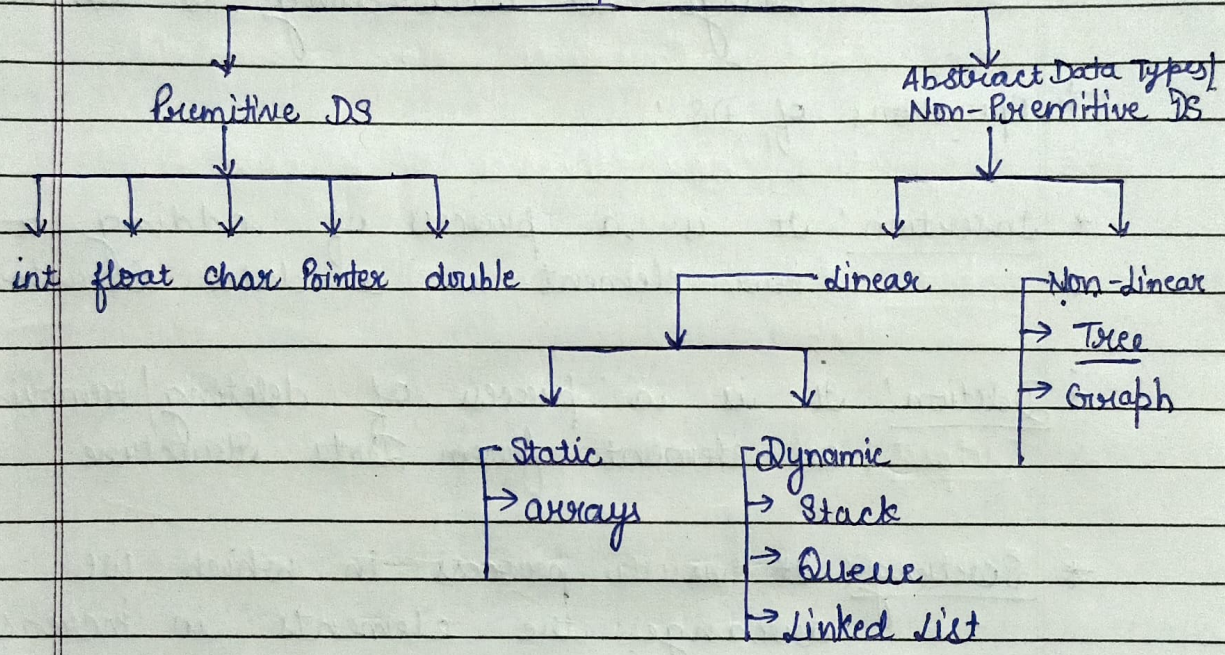
Data Structure

Data: Anything to given information is known as data. eg: Student's name

Structure: Representation of data.

Data Structure: It is a way of organizing data in such a way so that it can be used in an efficient way | (better way).

Types of DS



* **Premitive DS:** These are basic data structure and directly operated by machine instruction

* **Non Premitive / ADT:** These are drived from premitive DS.

- **Tree:** No cycle, Nodes are connected
- **Graph:** There is cycle when we connect nodes.

Date / /
Page No.

* Linear DS: In this we arrange data in a sequential way (ascending / decending order).

* Non Linear: In this data is arranged in an heriracial way/random way

* Static: Memory is fix. We can't change it again & again.

* Dynamic: Memory is not fix. We can change it according to our wish.

⇨ Operations of DS:

* Insertion: It is a process of adding a new element in a data structure.

* Deletion: It is a process of deleting/removing an element from Data Structure

* Sorting: It is a process in which we arrange the elements in increasing or decreasing order.

* Merging: It is a process when we combine two data structures into a single data structure.

* Traversing: In this we visit each element atleast one time.

* Searching: Finding / Searching an element in DS.

⇨ Importance of DS / Advantages

* Efficient Data Processing: When we store data efficiently to save the memory / storage.

* Algorithm Design: Follow steps in an efficient way.

* Memory Management: Utilizing memory properly.

* Scalability: Proper designing, maintaining of DS.

⇨ Disadvantages of DS:

* Complexity: Not all data types are complex but few are eg: tree, graph etc

* Memory Overhead: difficult that what we utilize where. Memory wastage also occur.

* Fixed Size: We can't change the size of arrays.

* Performance tradeoff: When we use wrong data type then this leads to decrease the performance of algorithm.

* Difficult in choosing right DS!

* Data Duplication: When we use 2 data types at a same time leads to data duplication.

⇒ Algorithm: Step by step instruction / process for solving any problem.
OR
It is well defined procedure that takes some input & produce some output.

⇒ Properties / characteristics of Algorithm:

1. Input: It should take 0 or more than 1 input.
2. Output: It must produce an output.
3. Finiteness: It must have finite number of steps.
4. Unambiguous: Each & every step must be clearly defined. No duplicacy
5. Effectiveness: Each & Every step must be followed

Date / /
Page No.

⇒ Different types of Algorithms:
* Selection: Decision making + Controle Statements
* Sequence: No controle statements & no decision making
* Iteration

□ Sequence eg: Add 2 no's.

Step 1 : Start
2 : Read a, b
3 : Sum = a+b
4 : display Sum
5 : Stop.

□ Selection eg: eligible for Vote:

Step 1 : Start
2 : Read age
3 : if age ≥ 18; go to step IV else Step V
4 : Write eligible for Vote
5 : Write not eligible for Vote
6 : Stop.

□ Steration Eg: fibonachi Series, Factorial.

Ques1 Write an algorithm to check whether a no. is even or odd.

Ques2 Write an algorithm to find greatest among 5 no's.

Ques3 Write an algorithm to find factorial of a no.

Solution

1. Step 1: Start
   2: Read a
   3: if a%2==0; go to step 4 else
   4: write even
   5: write odd.
   6: Stop.

2. Step 1: Start
   2: Read a, b, c, d, e
   3: if ((a>b) && (a>c) && (a>d) && (a>e)):
      write a is greater.
   4: else if ((b>a) && (b>c) && (b>d) && (b>e)):
      write b is greater.
   5: else if ((c>a) && (c>b) && (c>d) && (c>e)):
      write c is greater.
   6: else if ((d>a) && (d>b) && (d>c) && (d>e))
      write d is greater.
   7: else
      write e is greater.
   8: Stop

3. Step 1: Start
   2: Read a
   3: if (a==0) && (a==1):
      write 1
   4: else:
      while a>0:
      fact = 1
      fact = fact * a
      a--

5! Write fact.
6! Stop.

⇨ Flow chart:

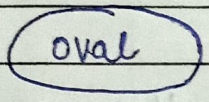
↳ Graphical representation of an algorithm.

Algorithm: time & memory should be less amount of

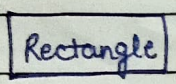
★ Time Complexity: Execution time of 1 Algo.

★ Space/Memory Complexity: Space that an algo take

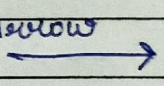
⇨ Symbols

1. 

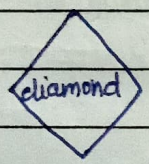
→ Start/Stop

2. 

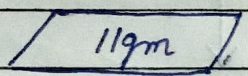
→ Process/operation

3. 

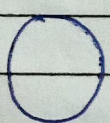
indicates the flow b/w steps

4. 

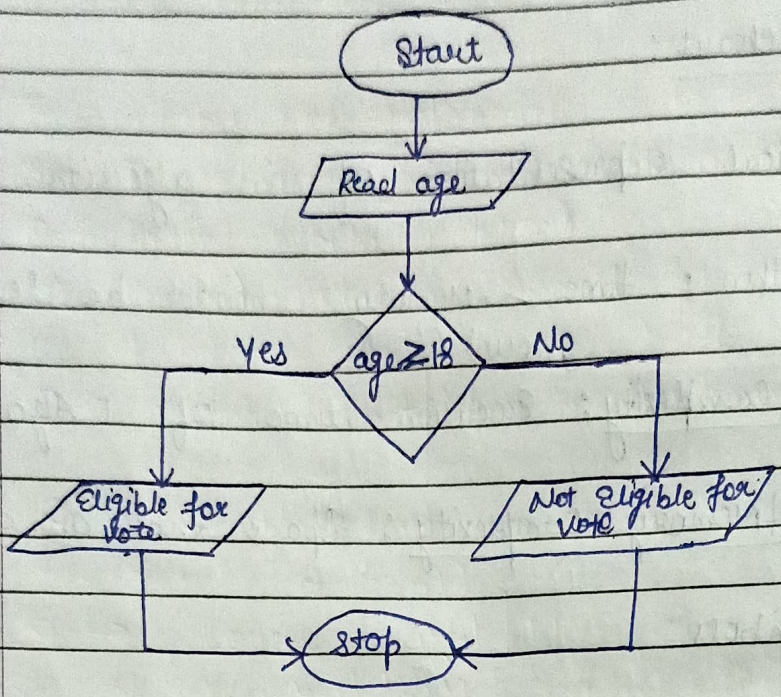
→ condition

5 

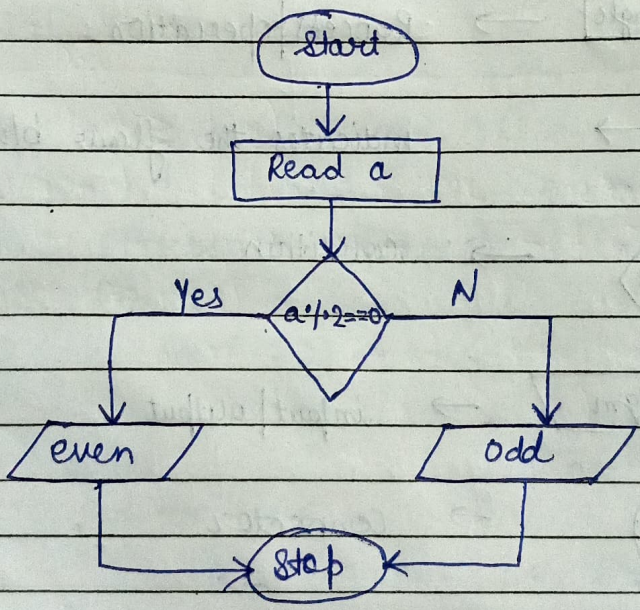
→ input/output

6. 
→ Connector

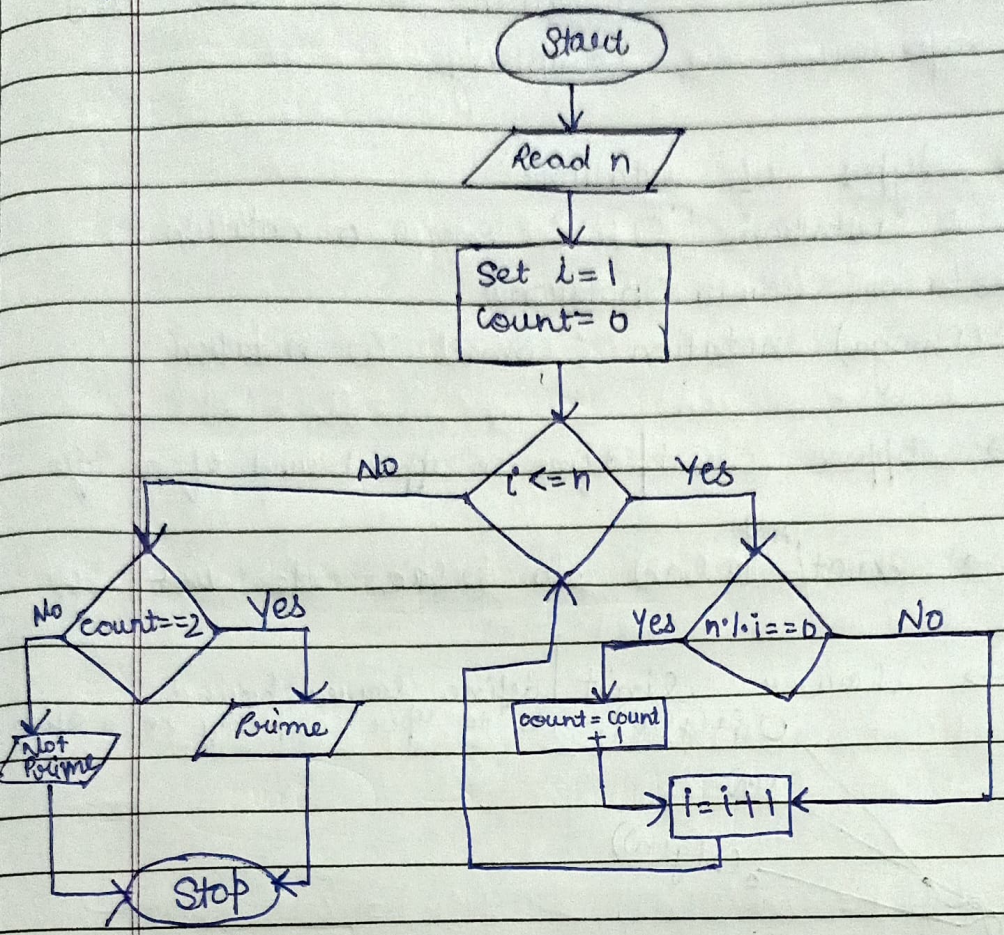
Eg.: Whether a person is eligible for vote or not.



eg. Make a flow chart to check whether a no. is even or odd.



eg whether a no. is prime or not.



⇨ Algorithm analysis :

Problem has more that one solution then which technique we are using to solve that particular problem is called as algo. analysis.

Analysis is done on two parameters

* Time Complexity
* Space Complexity

Two types of Algo. Analysis :

* Apriori analysis : Exact time & Space consume
* Apostercori analysis : Approx

2b Define algorithm complexity and its types along with algorithmic notations.

Ans Complexity in algorithms refers to the amount of resources (such as time or memory) required to solve a problem or perform a task. The most common measure of complexity is time complexity, which refers to the amount of time an algorithm takes to produce a result as a function of the size of the input. Memory complexity refers to the amount of memory used by an algorithm.

Types of complexity:

(i) Constant complexity: If the function or method of the program takes negligible execution time. Then that will be considered as constant complexity.

(ii) Logarithmic complexity: It imposes a complexity of O(log(N)). It undergoes

the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.

3 Linear Complexity: It imposes the complexity of O(N). It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.

4. Quadratic Complexity: It imposes the complexity of $O(n^2)$. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.

5 Exponential complexity: It imposes a complexity of $O(2^N)$, $O(N!)$, $O(n^k)$ . . .
For N elements, it will execute the order of the count of operations that is exponentially dependable on the input data size.

⇨ Algorithmic Notations:

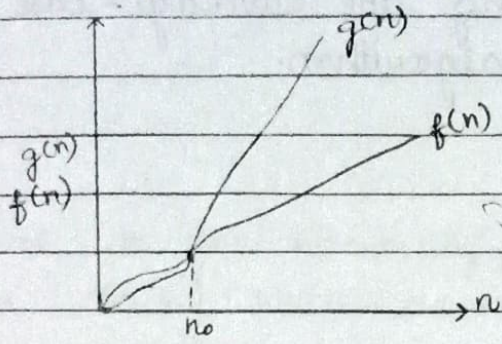
- Asymptotic Notation: It is a mathematical notation that are used to represent time complexity

Three Types of notations:
1. Big O notation, Little o notation
2. Omega notation, Little ω notation
3. Theta notation

Page No.
Date

- **Big O Notation** : Big - O notation represents the upper bound of the running time of an algorithm. By using big O- notation, we can asymptotically limit the expansion of a running time a range of constant factors above and below. It gives the worst case complexity of an algorithm. It is a model for quantifying algorithm performance.

O:

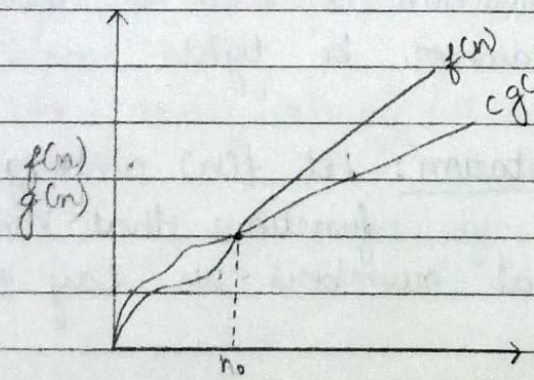


$O \rightarrow g(n) \geq f(n)$

$O(g(n)) = \{ f(n)$: there exist +ve constants c and no such that $0 \leq f(n) \leq cg(n)$ for all $n \geq no \}$

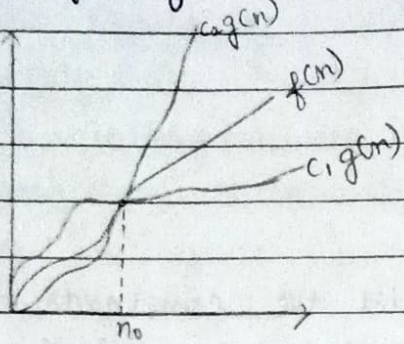
- **Omega Notation** : It represents the lower bound of the running time of an algorithm. Thus, it provides the best-case complexity of an algorithm. The execution time serves as a lower bound on the algorithms time complexity. It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

Ω:



$\Omega(g(n)) = \{f(n)$: there exist +ve constants c and n_0 such that $0 < f(n) \geq cg(n)$ for all $n \geq n_0\}$

- Theta Notation: Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used to analyzing the average-case complexity of an algorithm.



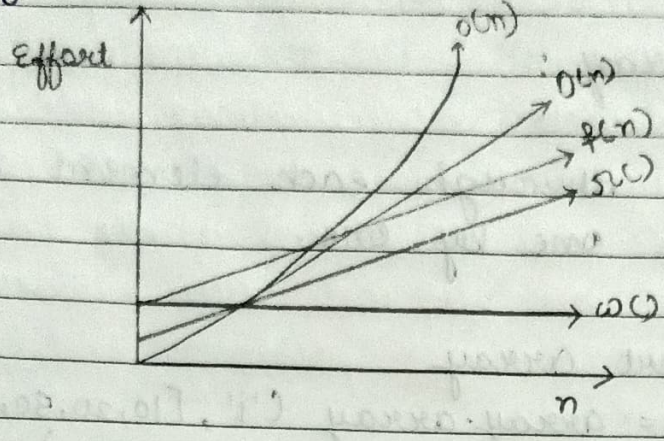
$\Theta(g(n)) = \{f(n)$: there exist +ve constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$

- Little o asymptotic notation: Big-O is used as a tight upper bound on the growth of an algorithm's effort (this effort is described by the function $f(n)$, even though, as written, it can also be a loose upper bound. "Little-o" ($o()$) notation is used to describe an upper bound that cannot be tight.

- Little ω asymptotic notation: Let $f(n)$ and $g(n)$ be functions that map +ve integers to +ve real numbers. We say that

Page No.:
Date:

$f(n)$ is $\omega(g(n))$ (or $f(n) \in \omega(g(n))$) if for any real constant $c > 0$, there exist an integer constant $n_0 \geq 1$ such that $f(n) > cg(n) \geq 0$ for every integer $n \geq n_0$.



⇒ Properties of Asymptotic notation

* Reflexive property : only O, Ω, Θ

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

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

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

* Symmetric property : Only theta shows Symmetric property

$f(n) = \Theta(g(n))$ if & only if $g(n) = \Theta(f(n))$

a = b

Date / /
Page No.

* Transpose Symmetry:
$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.
$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$.

* Transitivity property: All notation
If $f(n) = O(g(n))$ and $g(n) = O(h(n))$
then $f(n) = O(h(n))$

If $f(n) = \Omega(g(n))$ & $g(n) = \Omega(h(n))$ then
$f(n) = \Omega(h(n))$.

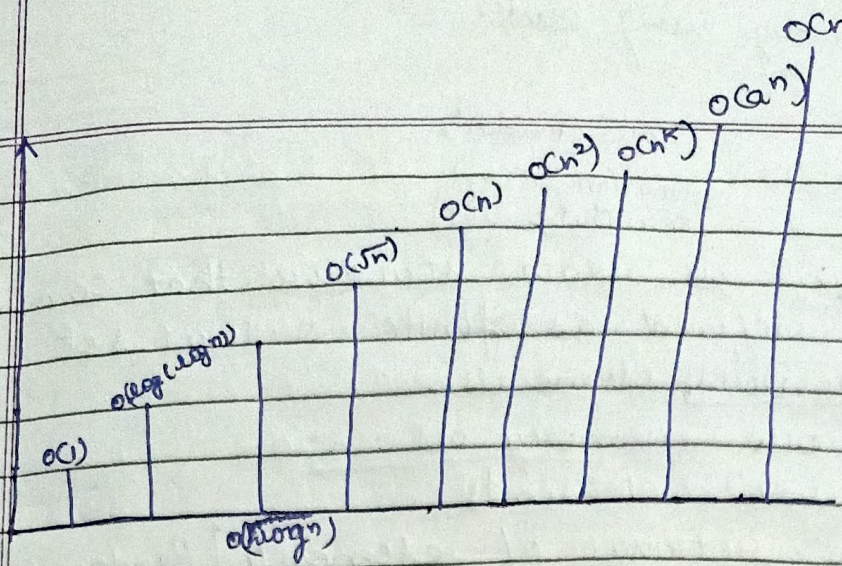
If $f(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n))$ then
$f(n) = \Theta(h(n))$.

If $f(n) = o(g(n))$ & $g(n) = O(h(n))$ then
$f(n) = o(h(n))$.

If $f(n) = \omega(g(n))$ & $g(n) = \omega(h(n))$ then
$f(n) = \omega(h(n))$

⇒ Types of Complexities:

1. Constant time complexity = $O(1)$
2. Logarithmic " " $= O(\log(\log(n))$, $O(\sqrt{\log n})$, $O(\log n)$.
3. linear time complexity = $O(\sqrt{n})$ and $O(n)$
4. Polynomial " " $= O(n^k)$ where $k =$ constant & $k > 1$
5. Exponential time complexity = $O(a^n)$ where $a > 1$.



Lab Inherit function: String Concatination → strcat
Kush part string da concat kran ge Strncat
strncat (Str1, Str2, 2) → index
Jine letter jodne a

⇒ Difference b/w Linear and Non-Linear DS.

Linear DS	Non-Linear DS
* The elements are arranged in a sequence such as array, stack or queue.	Elements are organized/ arranged in Random/ hierarchical order.
* In linear ds single level is involved.	Multiple levels are involved.
* Easy to implement	Difficult to implement
* Data elements can be traversed in a single run.	Cannot traversed in a single run.
* Memory is not used in an efficient manner.	Memory used in efficient manner

Ques3 What do you mean by array? How to declare an an array? Write down all the operations of array.

Ans **Array:** It is a linear data structure that stores a collection of items of same data type in contiguous memory location. Each item in an array is indexed starting with 0. We can directly access an array element by using its index value.

⇨ Declaration of Array:

Arrays can be declared in various ways in different languages, we declare array in python.

→ Using array module:

```
Import array
arr = array.array('i',[1,2,3,4,5])
```

Here 'i' specifies the type of element in array.

→ Using Numpy Module

```
Import numpy as np
arr = np.array([1,2,3,4,5])
```

⇨ Operations In array :

1. Traversing: Going through each element in the array one by one.

For example: Import array

```
arr = array.array('i', [10,20,30,40])
for i in range(len(arr)):
    print(arr[i])
```

2. Insertion: Adding a new element to the array

For example:

```
arr = [10,20,30,40]
arr.append(50)
```

3. Deletion: Removing the element from array

For example:

```
arr = [10,20,30,40]
arr.pop(1)  # Remove element at index 1
```

4. Searching: Locating an element in the array.

For Example:

```
arr = [10,20,30,40]
found = False
for i in range(len(arr)):
    if arr[i] == 30:
        found = True
        break
```

5. Update : Changing the value of element at a particular index.

For Example : arr = [10, 20, 30, 40]

arr [3] = 50 #, change the element at index 3 to 50.

Array

- One Dimensional Array
- Multi-Dimensional Array
  - Two Dimensional
  - Three Dimensional

- One Dimensional Array : Single Row and multiple Columns.

eg: array = | 10 | 20 | 30 | 40 | 50 |

→ 2 ways to import array module:

1. import array – This will import the entire array module.

2. from array import * – This will import all class, objects, variables etc from array module.

3. import numpy

→ Creating & Initializing 1-D Array:

```
import array as arr
array_name = arr.array('type_code', [elements])
```

eg
```
import array as arr
marks = arr.array('i', [10,20,30,40,50])
```

- Two-Dimensional Array: List of lists which represent a table like structure with rows & colums

import numpy as np.
array name = np.array([[elements], [elements]])

eg: a = np.array([[1,2,3],[1,2,3]])

- Three-Dimensional Array:

import numpy as np
array name = np.array([[[elements], [elements]],
[[elements], [elements]]])

OR

array name = np.arange(1,25).reshape(2,3,4)

output: [[[[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]],

[[13, 14, 15, 16],
[17, 18, 19, 20],
[21, 22, 23, 24]]])

*

→ Numericals on Array: 1D array.

- Address of the element at k^{th} index.
- a[k] = base address + Size of 1 element * [No. of elements before the element which is given (Given index bound - lower bound)]

eg 1. A = [0, 1, 2, 3 ------ 100]
Size = 4 Byte
Base address = 1001
Find the address of the element that is present on 50th index.

⇒ 1001 + 4(50 - 0) = 1001 + 4(50)
= 1001 + 200 = 1201 Ans

eg 2 Consider the linear array A [16 : 30]. If base address is 100 and space required to store each element is 4 words then the address of A[27] is given by.

⇒ A[27] = 100 + 4(27 - 16)
= 100 + 4(11)
= 100 + 44 = 144 Ans

*

→ Numericals in 2D Array:

Two Methods → Row Major Storage
→ Column Major Storage

B - Base Address W = Storage size of an element
Lr - Lower bound of row
Lc - Lower bound of Column
M - No. of rows in matrix
N - No of columns in matrix
i = row subscript of element whose add to be found
j = Column " " " " " " " " eg- A[i][j]

Numericals of 2D Array

Date / /
Page No.

1. Row Major :

Address of A[i][j] = B + W * [N * (i - Lr) + (j - Lc)]

eg A[2][1] = 104 + 2 * [4 * (2 - 0) + (1 - 0)]

= 104 + 2 * [8 + 1]

= ~~106 (9)~~ = 104 + 2(9)

= 104 + 18 = 122 Ans

2. Column Major :

Add of A[i][j] = B + W * [(i - Lr) + M * (j - Lc)]

eg: A[1][2] = 104 + 2 * [(1 - 0) + 3(2 - 0)]

= 104 + 2 * [1 + 6]

= 104 + 2 * (7)

= 104 + 14 = 118 Ans

	0	1	2	3
0	5	7	4	3
1	1	0	9	8
2	6	2	1	4

R (LB UB) C (LB UB)

Ques An array M[1:10, 1:10] required 2 bytes of the storage for each element and the beginning location is 500 so determine the location of M[5][6].

⇒ Row major : M[5][6] = 500 + 2 * [10(5 - 1) + (6 - 1)]

= 500 + 2 [40 + 5]

= 500 + 2(45)

= 500 + 90

= 590 Ans.