Complexity of an algorithum

The complexity of an algorithum is the function $f(n)$ which gives running time or storage space requirement of the algorithum in terms of size of input data (n).

Time complexity:- Represents total amount of time needed to run the algorithum completely.

It is denoted by $t(n)$.

If one step of an algorithum needs C time to run and there are n terms then total time is,

$$t(n) = C * n = Cn$$

Space Complexity:-

Represent the amount of memory space required by algo. It is sum of following two:-

- A fixed part memory independent of size of problem.

- Variable part of space, which dependent on size of problem.

→ Execution time case :-

Time taken by algo. to execute in different cases. It is used to compare many algo.

- Worst case :-

In this case algo. takes max time to execute, which it can take.

It provides upper bound to the running time. If it is low that means algo is better, as in worst cases it will take less time.

- Average case :-

This shows average execution time to run algo.

- **Best case :-**

It shows lowest time to run/execute the algo.

It provides lower bound to the execution time of algorithum.

Q. Calculate total time to execute given code and it's time complexity.

i> `for(i=0;i<n;i++)` → c_1
`a[i]=0` → c_2
] takes for one time to run.

→ here, for loop runs $(n+1)$ times
So, time taken is $c_1(n+1)$

statement inside for loop runs 'n' times. So, it takes

$c_2 n$ time.

hence,

$$T(n) = c_1(n+1) + c_2 n$$
$$= n(c_1+c_2) + c_1$$

NOTE :- While calculating Time complexity we neglect terms with lower power of n.

So, in previous example

$$T(n) \approx O(n)$$

ii)
```
Sum = 0;                        c1
for(i=0; i<n; i++)              c2
    for(j=0; j<n; j++)          c2
        Sum += a[i][j];         c3
```

→ First step sum=0; takes c_1 time

first for loop runs (n+1) times so,

$c_2(n+1)$

second loop runs (n+1) times, each times first for loop runs. so, it runs

$n \cdot (n+1)$ times so,

$c_2 \cdot n \cdot (n+1)$

last step runs $n \cdot n$ times. so,

$n \cdot n \cdot c_3$

So,

$$T(n) = c_1 + (n+1)c_2 + n(n+1)c_2 + n^2 c_3$$

which is in range of n^2, so,

$$T(n) \approx n^2$$

=

⇒ Asymptotic notation:- Notation to represent complexity in mathematical term:-

→ Thita (Θ) notation:-

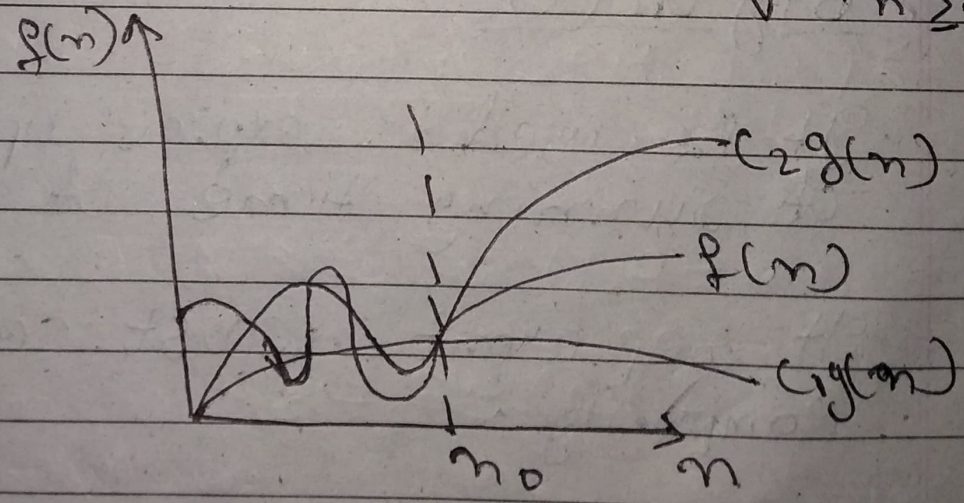
Represent upper and lower bound of run time of any algorithum. Basically average case of algo.

mathematical notation of (Θ):-

f and g non(-ve) fun^{n}.

$\Theta(g(n)) = \{ f(n) \mid f$ is non(-ve) fun^{n} such that there exist ($\exists$) +ve content c_1, c_2, n_0, such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall\ n \geq n_0$$



Properties:-

- $f(n) \in \Theta(g_1)$ and $h(n) = \Theta(g_2)$
  $f(n) + h(n) \in \Theta(g_1 + g_2)$

- $g = g_1 = g_2$

then $f(n) + h(n) \in \Theta(g)$

NOTE:- It is also right:-

$f = \Theta(g)$

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

$f \in \Theta(g)$

all 3 are same.

we can't write :-

$[\Theta(g) = f]$ it is wrong.

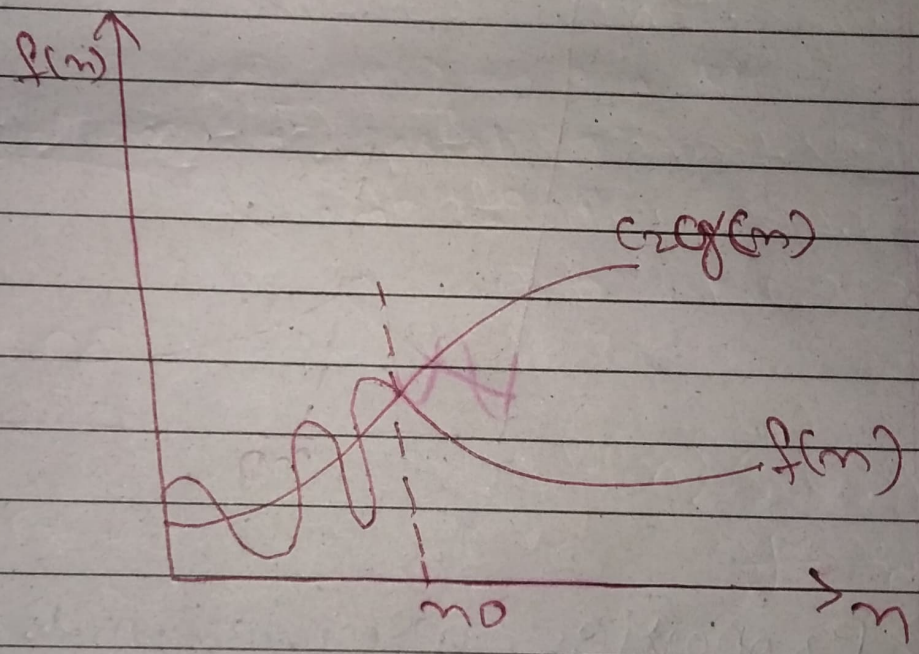
→ Big Oh (O) :-

It is used to express upper bound of running time of algorithum.

It represent worst case time complexity.

mathematical representation

$O(g(n)) = \{f(n) \mid f(n)$ is non(ve) function there exist (+ve) constant 'c' and 'no' such that $f(n) \leq c_2 g(n) \; \forall \; n \geq n_0\}$



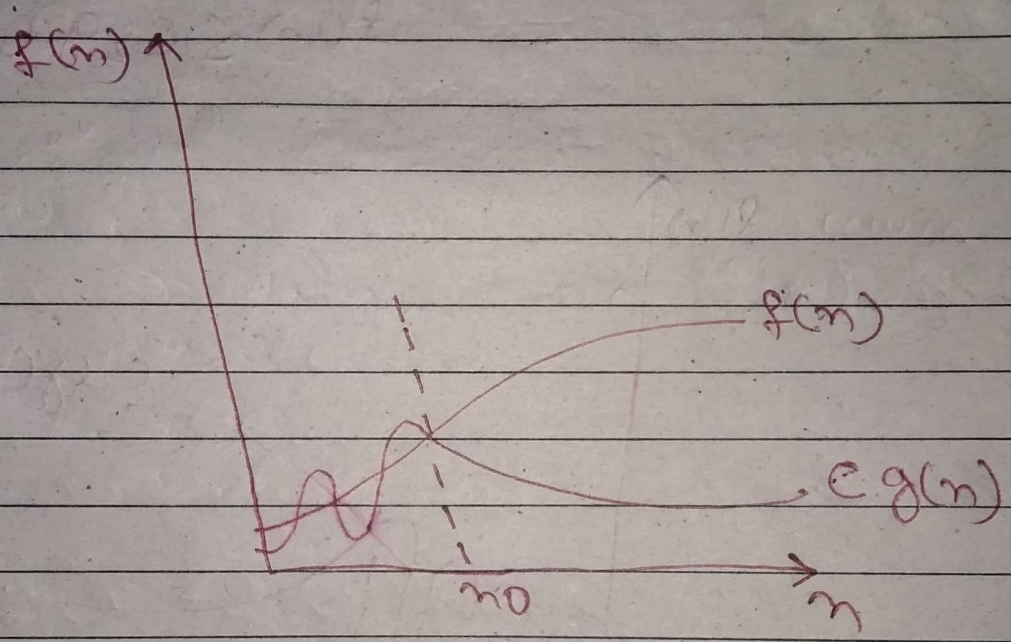
→ Omega notation (Ω) :-

It is used to express lower bound of an algorithum's running time.

It represent best time complexity.

mathematical representation :-

$\Omega(g(n)) = \{f(n) \mid f(n)$ is non (+ve) function, there exist two (+ve) constant C and n_0, such that $f(n) \geq C(g(n)$ $\forall \quad n \geq n_0\}$



→ Properties associated with asymptotic notation :-

- $f \in \Theta(g) \Rightarrow f \approx g$
- $f \in O(g) \Rightarrow f \leq g$
- $f \in \Omega(g) \Rightarrow f \geq g$
- $f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n))$ and $f = \Omega(g(n))$

→ Transitivity law :-

- $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$ then $f(n) = \theta(h(n))$

- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$

same for Ω.

→ Reflexive law :-

- $f(n) = \theta(f(n))$ same for O & Ω.

→ Symmetry law :-

- $f(n) = \theta(g(n))$ iff $g(n) = \theta(f(n))$

Q. prove that :-

1. $f(n) = 10n^3 + 5n^2 + 7 \in \theta(n^3)$

→ If given function is of 'θ' function, implies,

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

$\Rightarrow 5n^3 \leq f(n) \leq n^3(10+5+17)$

here, note that for c_1 choose smallest possible value from coefficient or choose your own smallest +ve integer, for c_2 calculate sum of all coefficient or choose other big number.

for $g(n)$ choose maximum order 'n' value from $f(n)$, in this case n^3.

$\Rightarrow 5n^3 \leq f(n) \leq 32n^3$

here, $c_1 = 5$ and $c_2 = 32$

now, use hit and trail method to calculate value of n_0 by changing value of n, such that it satisfy the equation of given asymptotic notation. (for $n_0 = n$, after that all value of n., $f(n)$ must follow mathematical eq. of that notation)

for, $n = 1$

$\Rightarrow 5 \times 1^3 \leq 32 \leq 32$

$\Rightarrow 5 \leq 32 \leq 32$

for, $n = 2$

$\Rightarrow 5 \times 2^3 \leq 117 \leq 256$

$\Rightarrow 40 \leq 117 \leq 256$

for $n = 3$

$\Rightarrow 5 \times 3^3 \leq 332 \leq 864$

So, as we see, equation is correct after $n = 1$.

So, $n_0 = 1$.

hence, $C_1 = 5$, $C_2 = 32$ and $n_0 = 1$, given function $f(n) \in \Theta(n^3)$.

2. $f(n) = 10n^3 + n \log n \in \Theta(n^3)$

$\Rightarrow$ According to mathematical eq.

$C_1 g(n) \leq f(n) \leq C_2 g(n)$

$\Rightarrow 10n^3 \leq f(n) \leq 11n^3$

where, $c_1 = 10$ and $c_2 = 11$.

now, for $n = 1$

$f(n) = 10 \times 1^3 + 1 \cdot \log 1$

$= 10 + 0$

$= 10$

$\Rightarrow 10 \times 1^3 \leq 10 \leq 11 \times 1^3$

$\Rightarrow 10 \leq 10 \leq 11$

$=$ true

for, $n = 2$

$f(n) = 10 \times 2^3 + 2 \times \log 2$

$= 80 + 2 \times 0.6$

$= 81.2$

$\Rightarrow 10 \times 2^3 \leq 81.2 \leq 11 \times 2^3$

$\Rightarrow 80 \leq 81.2 \leq 88$

$=$ true

as we can see for $n_0 = 1$, $f(n) \in O(n^3)$

Q 3. $f(n) = 2 + \frac{1}{n} \in \Theta(1)$ → class of constant.

→

$$2 \leq 2 + \frac{1}{n} \leq 3$$

for $n = 1$

$$2 \leq 3 \leq 3$$

for $n = 2$

$$2 \leq 2.5 \leq 3$$

for $n = 3$

$$2 \leq 2.33 \leq 3$$

hence, for $n = 1$, $f(n) \in \Theta(1)$.

Q. If $f(n) = n$ and $g(n) = \log n^2$ then find what relation b/w them, is it Θ, O or Ω.

⇒ To solve such type of problem, you have to use hit and trail method, change value of n and check value of $f(n)$ and $g(n)$

If $g(n) \geq f(n) \Rightarrow O$

If $g(n) \leq f(n) \Rightarrow \Omega$

If some times $f(n)$ is grater and some time $g(n)$ is grater then it is Θ. But for 'Θ' you have to check for wider value range of n. as it might be the case that some value you have choosen for n is not $\leq n_0$ but less than that.

→

	$f(n) = n$	$g(n) = \log n^2 = 2\log n$
$n=1$	1	$2 \times 0 = 0$
$n=2$	2	$2\log 2 = 0.6$
$n=3$	3	$2\log 3 = 0.9$
$n=5$	5	$2\log 5 = 1.39$
$n=10$	10	$2\log 10 = 2$
$n=100$	100	$2\log 100 = 4$

as we can see from these results

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

Q. $f(n) = 2^n$ $g(n) = 3^n$

→

	$f(n) = 2^n$	$g(n) = 3^n$
$n = 1$	2	3
$n = 3$	8	27
$n = 5$	32	243
$n = 10$	1024	59,049

Q2, you can see for $n_0 = 1$,

$f(n) \leq g(n) \Rightarrow f(n) = O(g(n))$

Q. $f(n) = 2^n$ $g(n) = n^2$

	$f(n) = 2^n$	$g(n) = n^2$
$n = 1$	$2^1 = 2$	$1^2 = 1$
$n = 2$	$2^2 = 4$	$2^2 = 4$
$n = 3$	$2^3 = 8$	$3^2 = 9$
$n = 4$	$2^4 = 16$	$4^2 = 16$
$n = 5$	$2^5 = 32$	$5^2 = 25$
$n = 6$	$2^6 = 64$	$6^2 = 36$
$n = 7$	$2^7 = 128$	$7^2 = 49$
$n = 10$	$2^{10} = 1024$	$10^2 = 100$

as you can see, for $n \geq 4$

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

→ Array representation :-

In one dimentional array, array is represented in one way, linearly but in case of 2-D array, it can be represented in 2 different ways.

i) Row major order ii) Column major order.

→ Row major order :- (RMO)

In row major order, elements are stored row-wise in the memory in one dimensional form.

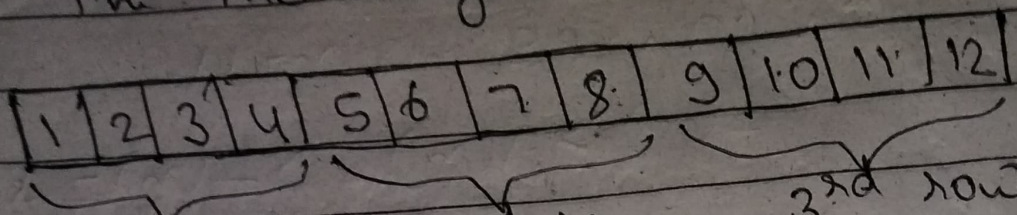
ex.-

$a[3][4]$ =

1	2	3	4
5	6	7	8
9	10	11	12

3×4

in memory it is like:



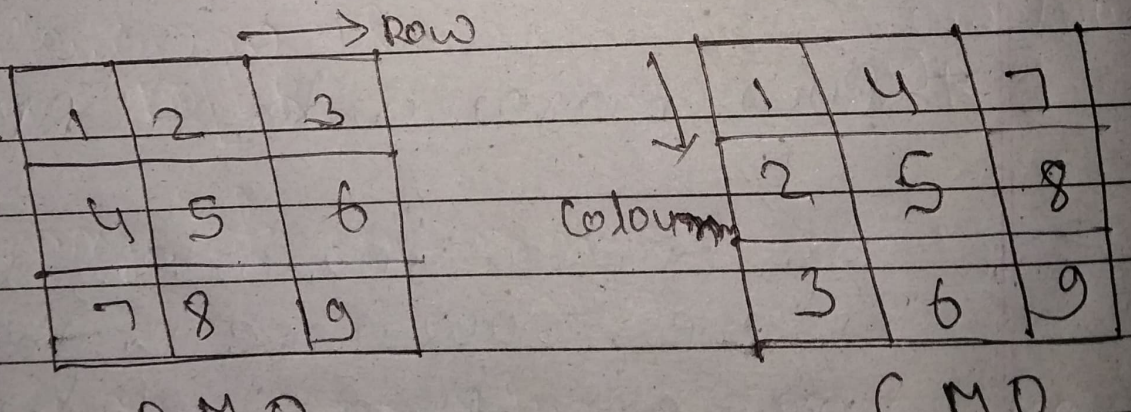
By default arrays are RMO.

→ Column major order:-(CMO)

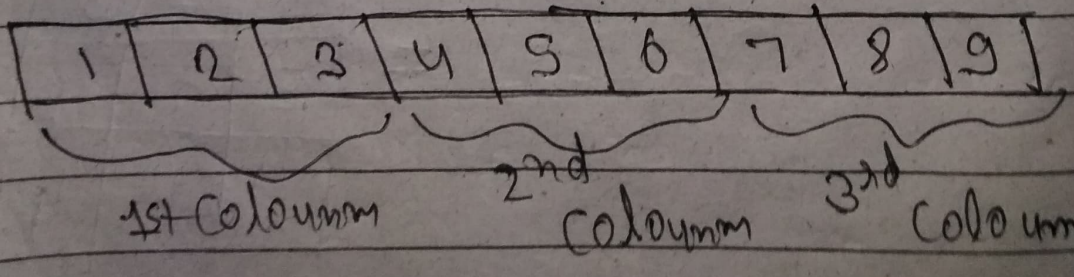
In column major order, all the elements of first column are printed, then second column and so on.

ex.-

A matrix is RMO and CMO



in memory



→ Address calculation of elements in 1-D array :-

In case of 1-D array for RMO and CMO, for both formula is same.

add. of a[i] = base add + i x w

where, base add is add of first element of array a[0].

i is index of element.
w is byte size of element.

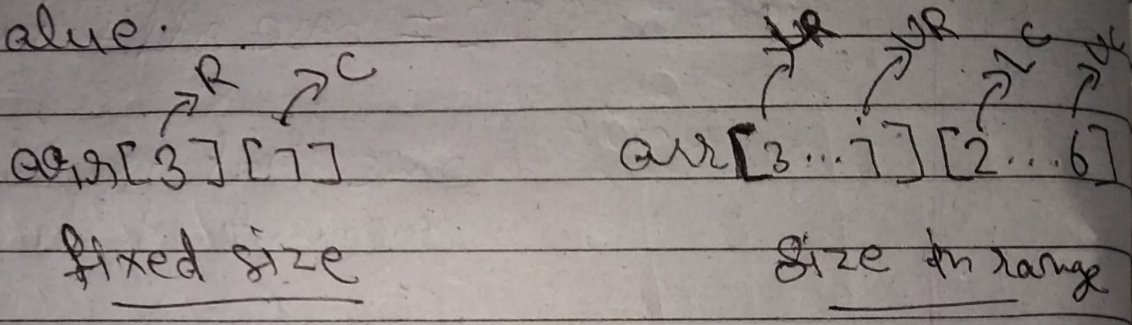
Q. Calculate add. of element A[2] if base add is 2000 and w = 2.

→

add. of A[2] = 2000 + 2 x 2 = 2004

→ Address calculation of 2-D array:

In case of 2-D array there are many variations in formula as, for RMO and CMO, formula will change, also in case of 2-D array size can be fixed or a range of value.



In case where size is in range, we assume, range of row as

[lower row value .... upper row value]
[LR ... UR]

and

range of column as

[lower column value ... upper column value]
[LC ... UC]

In case of fixed size we assume lower row value(LR) and lower column(LC) value as 'zero' and given sizes as

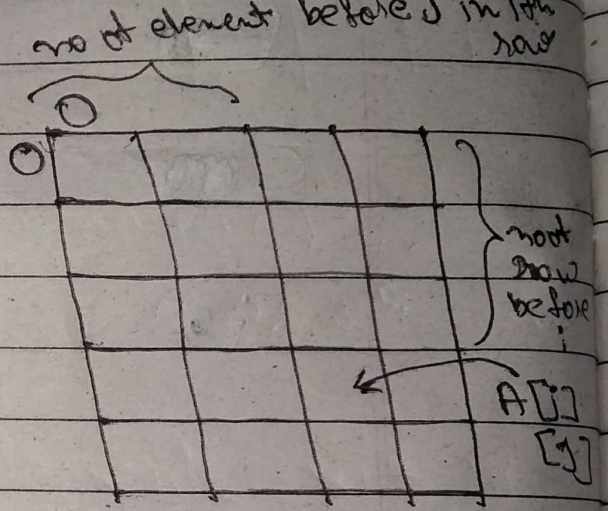
Upper row (UR) and Upper column (UC).

now, Let's assume we have to calculate address of A[i][j] then

for fixed size array	for size as range	
R	UR-LR+1	Total no. of rows
C	UC-LC+1	Total no. of columns
i	i-LR	no. of row before ith row (RMO) OR no of element before ith row (CMO)
j	J-LC	no of elements before jth column (RMO) OR no of column before jth column (CMO)

So to formulate this
To make formula from these information we have to first visualise 2-D array. How many elements we have to cross from base-add element to reach the given element, if we get that, we get our answer.

- RMO :- In this array A[5][4], we have to find, add of A[i][j], i=3, j=2



now, we can see,

no of elements in rows before ith row = no of row before ith row × Total no of column

$$= (i-LR)(UC-LC+1)$$

no of element before jth column in that row = $j-LC$.

So, final formula is,

$$\text{add. of } A[i][j] = B + \left[(i-LR)(UC-LC+1) + j-LC\right] \times w$$

where,

B is base add of A[0][0]

w is byte size of one element in array.

of a fixed size

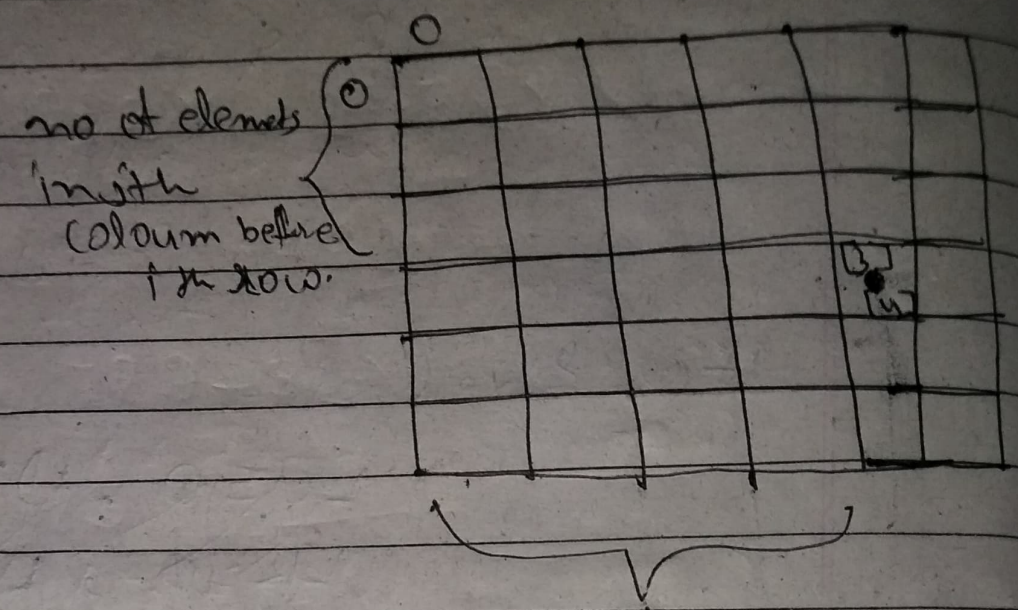
[add of A[i][j] = B + [i x C + j] x w]

NOTE:- If indexing of array is started from [1][1] then the formula becomes,

add of A[i][j] = B + [(i-1) x C + (j-1)] x w

- CMO:-

Let's assume we have an array A[5][6] and we have to find add of A[3][4].



As we know in CMO, elements starts to insert column by column.

So no of elements in, columns before jth column is

no of column before jth column x total no of row

$$= (j - LC) \times (UR - LR + 1)$$

no of elements in jth column before ith row - $(i - LR)$

So, final formula is

add of A[i][j] = B + [(j - LC) × (UR - LR + 1) + (i - LR)] × w

for fixed size -

add of A[i][j] = B + [j × R + i] × w

If indexing starts from [1][1] then in case of fixed size

add of A[i][j] = B + [(j - 1) × R + (i - 1)] × w

Q. An array A[-15...10, 15...40] requires one byte of storage. If beginning location is 1500 determine the location of x[15, 20].

→ base address = 1500

w = 1

i = 15 LR = -15 UR = 10

j = 20 LC = 15 UC = 40

Total no of rows = UR - LR + 1 = 26

Total no of column = UC - LC + 1 = 26

RMO:-

add of A[15][20] =

$1500 + [(15+15) \times 26 + 5] \times 1$

$= 1500 + [30 \times 26 + 5] \times 1$

$= 1500 + [780 + 5] \times 1$

$= 1500 + 785$

$= 2285$

CMO:-

~~add of~~ A[15][20] =

$1500 + [5 \times 26 + 30] \times 1$

$= 1500 + [130 + 30] \times 1$

$= 1660$

Q. A[30][20], w=8 bytes, Find location of A[5][10] if A[4][5] is stored at 3000.

→ R = 30, C = 20, w = 8