Lec-9

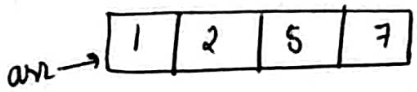
Arrays-3

12/Feb/2023

⊙ 2-D arrays-

1-D arrays

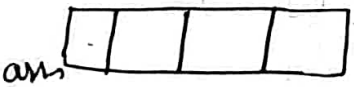
↳ `int arr[] = {1,2,5,7};`



`int a = 5`

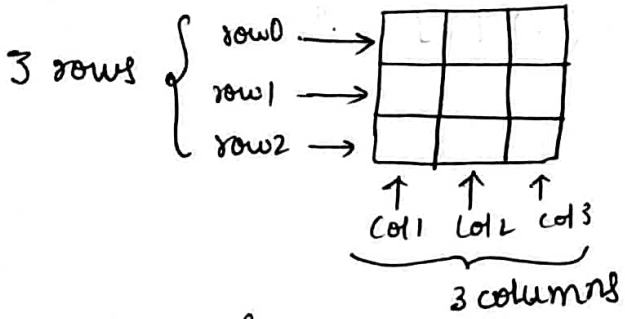
a [5]

`int arr[4];`



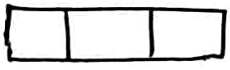
← garbage values.

2-D array → eg → Tic-Tac-To

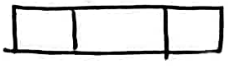


→ How can we make this with array?

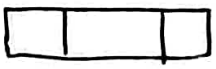
(1-D Array) → 0 Row →



1 Row →



2 Row →



3 rows → 3 1-D arrays.

But suppose we want to make 1000 row array then we have to create 1000 1-D arrays. But it's not convinient. So

So that's why 2-D Arrays are important.

How to create 2-D Array →

```
int arr[1000][1000];
```

(1000 ← no. of columns; 1000 ← no. of rows)

```
int arr[4] = {1,2,3,4};
```

inside memory →

1	2	3	4

```
int arr[3][3];
```

Inside memory

5	7	9
12	4	2
1	6	7

× no, the elements are not stored in 2-D Array inside memory.

It is just for visualization.

In memory 2-D anay also stored in contigous way. (linearly)

```
int arr[3][3];
```

→

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

Now, how the values are mapped? How to access?

- How to access an element -

```
arr[1][0] = 12
```

(1 ↑ row, 0 ↑ col)

arr	0	1	2
0	5	7	9
1	12	4	2
2	1	6	7

arr[2][2] = 7

arr[1][1] = 4

arr[0][2] = 9

formula = c * i + j

where c = no. of col
i = i^{th} row
j = j^{th} col

	0	1	2
0	2	4	6
1	8	9	10
2	13	15	17

in memory →

0	1	2	3	4	5	6	7	8
2	4	6	8	9	10	13	15	17

arr[2][1]

i = 2, j = 1

formula = c * i + j

= 3 × 2 + 1 = 7

arr[1][2]

i = 1, j = 2

= c * 1 + 2 = 3 × 1 + 2 = 5

But that doesnot mean that if you want to access 15 in this example you will access like this → arr[10] ——> This is wrong.

- **Declaration –**

```
int arr[3][3];
```

- **Initialization –**

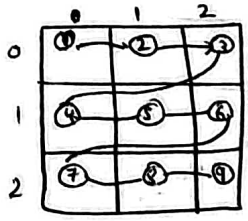
```
int brr[3][3] = { {1,2,3}, {4,5,6}, {7,8,9}};
int arr[2][2] = {{1,2},{3,4}};
```

	0	1
0	1	2
1	3	4

- **Taking input and printing the array →**

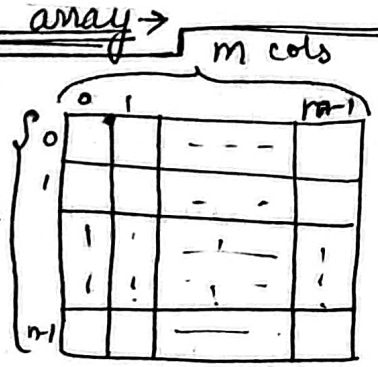
```
int arr[3][3];
```

row-wise access this array.



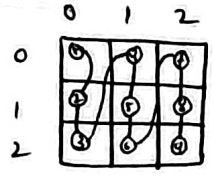
(0,0) (0,1) (0,2)
0^{th} row → 0^{th} col, 1^{st} col, 2^{nd} col
(1,0) (1,1) (1,2)
1^{st} row → 0^{th} col, 1^{st} col, 2^{nd} col
(2,0) (2,1) (2,2)
2^{nd} row → 0^{th} col, 1^{st} col, 2^{nd} col

Imp.



n rows, m columns,
⇒ indexes → 0 – (n-1) rows
0 – (m-1) cols.

column-wise access –



(0,0) (1,0) (2,0)
0^{th} col → 0^{th} row, 1^{st} row, 2^{nd} row
(0,1) (1,1) (2,1)
1^{st} col → 0^{th} row, 1^{st} row, 2^{nd} row
(0,2) (1,2) (2,2)
2^{nd} col → 0^{th} row, 1^{st} row, 2^{nd} row.

To print rowwise →

```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        cout << brr[i][j] << " ";
    }
    cout << endl;
}
```

(i → row, j → col)

And to print column-wise → The only change is

```
cout << brr[j][i] << " ";
```

Row-wise input -

```
int arr[3][3];
for (int i=0; i<3; i++){
    for(int j=0; j<3; j++){
        cin>> arr[i][j];
    }
    cout << endl;
}
```

col-wise input - The only difference is

```
cin >> arr[j][i].
```

Questions -

1. Row-sum print -

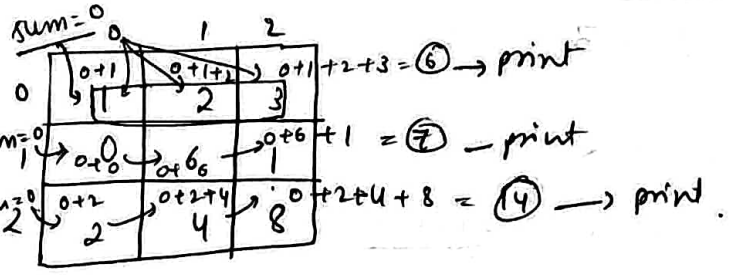
```
int main(){
    int arr[3][3];
    printRowWiseSum(arr, 3);
}
void printRowWiseSum(int arr[][3], int n)
{
    for (int i=0; i<3; i++){
        int sum=0;
        for(int j=0; j<3; j++){
            sum += arr[i][j];
        }
        cout << sum << " ";
    }
}
```

i/p →

1	2	3	→ 6
4	5	6	
7	8	9	

output → 6 15 24

```
int arr[][3][3];
```
↑ *Except this first all must have bounds.



2. Column-wise sum →

```
int main(){
    int arr[3][3] = _ _ _ ;
    printColWiseSum(arr, 3);
}
```

```
void printColWiseSum(int arr[][3], int n){
    for(int i=0; i<3; i++){
        int sum = 0;
        for(int j=0; j<3; j++){
            sum += arr[j][i];
        }
        cout << sum << " ";
    }
}
```

3. Search an element -

~~int arr[][]~~

return true or false.

i/p

1	2	3
4	5	6
7	8	9

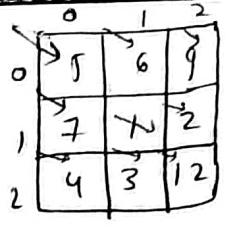
key = 2

o/p => True.

It does not matter if we search row-wise or col-wise.

```
~~int~~ bool findKey(int arr[][3], int rows, int col, int key){
    for(int i=0; i<row; i++){
        for(int j=0; j<col; j++){
            if(arr[i][j] == key)
                return true;
        }
    }
    return false;
}
int main(){
    int arr[][3] = {{1,2,3},{4,5,6},{7,8,9}};
    if(findKey(arr, 3, 3, 2))
        cout << "True";
    else
        cout << "False";
}
```

⑥ Max / Min element in a 2-D array →



~~max~~ = INT_MIN; 5 6 9 (12)

~~min~~ = ~~INT_MAX~~; 5 (1)

```
int getMax (int arr[][3], int rows, int cols){
    int maxi = INT_MIN;
    for (int i=0; i<rows; i++){
        for (int j=0; j<cols; j++){
            if (arr[i][j] > maxi){
                maxi = arr[i][j];
            }
        }
    }
    return maxi;
}
int getMin (int arr[][3], int rows, int cols){
    int mini = INT_MAX;
    for (int i=0; i<rows; i++){
        for (int j=0; j<cols; j++){
            if (arr[i][j] < mini)
                mini = arr[i][j];
        }
    }
    return mini;
}
```

⑦ Transpose of a Matrix

i/p →

2	4	6
1	3	5
9	11	13

o/p →

2	1	9
4	3	11
6	5	13

2 → arr[0][0]
4 → arr[0][1]
6 → arr[0][2]
(i: 0, j: 2)

2 → arr[0][0]
4 → arr[1][0]
6 → arr[2][0]
(i: 2, j: 0)

i and j swap.

⇒ swap(arr[i][j], arr[j][i])

```
void transpose (int arr[][3], int rows, int cols){
    for (int i=0; i<rows; i++){
        for (int j=0; j<cols; j++){
```

swap(arr[i][j], arr[j][i]);

}

}

}

i/p →

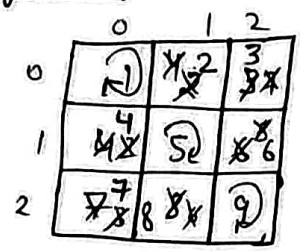
1	2	3
4	5	6
7	8	9

o/p →

1	2	3
4	5	6
7	8	9

But we did not get transpose

Let's dry run -



That means elements are swapping two times and they end up reaching their original place.
That's why we are not getting transpose.

That means we have to change loop's stoping condition.

```
void transpose(int arr[][3], int r, int c){
    for(int i=0; i<r; i++){
        for(int j=0; j<i; i++){
            swap(arr[i][j], arr[j][i]);
        }
    }
}
```

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

1 → (0,0) 4 → (1,0)
2 → (0,1) 5 → (1,1)
3 → (0,2) 6 → (1,2)
7 → (2,0)
8 → (2,1)
9 → (2,2)

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

1 → (0,0) 4 → (0,1) 7 → (0,2)
2 → (1,0) 5 → (1,1) 8 → (1,2)
3 → (2,0) 6 → (2,1) 9 → (2,2)

suppose 4x3 matrix →

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

4x3

→

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

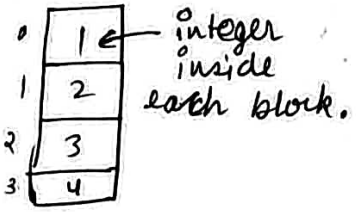
Some Famous Questions -

- Rotate matrix
- Spiral print
- wave print
- zig-zag print

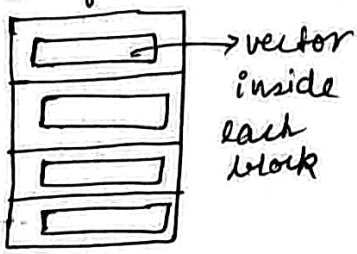
⊙ 2-D Vectors -

Creation → `vector < vector < int > > arr;` (int: datatype)

vector of int



vector of vector



Declaration → `vector < vector < int > > v;` (outer vector, inner vector, datatype, variable name)

A 3×4 2-D vector →

```
vector < vector < int >> arr (3, vector <int> (4));
```

(3: row)

Code -

```
vector < vector < int >> arr;
vector <int> a {1,2,3};
vector <int> b {4,5,6};
vector <int> c {7,8,9};
arr.push_back (a);
              (b);
              (c);
// Print the 2-D vector.
for (int i = 0; i < arr.size(); i++){
    for (int j = 0; j < arr[0].size(); j++){
        cout << arr[i][j] << " ";
    }
    cout << endl;
}
```

(arr.size(): size of row; arr[0].size(): size of col.)

arr[0].size() → works fine when no. of columns are equal in each row.

But when no. of columns are not equal we must use

arr[i].size()

vector<vector<int>> arr(rows, vector<int>(col, 0));

- rows: no. of rows or size of outer vector
- vector<int>(col, 0): initialized inner vector
- 2-D vector of integers.
- Initialize outer vector with this.
- col: size of inner vector.

~~vector<vector<int>>~~

vector<int> a(5, 0); → initialized each value with 0.

vector<vector<int>>(5, vector<int>(5, -8));

H.W
- → Colwise Sum
- → Other ways to find Transpose
- → Rotate by 90°

- → Sort 0's, 1's and 2's 1-D array.
- → Move -ve element to one side of array.
- → find duplicate element.
- → find missing element
- → find first repeating element
- → find common element in 3 arrays.
- → factorial of large no.
- → Spiral print
- → Wave print